(4)

The Rhet Programmer's Guide (For Version 15.25)

Bradford W. Miller

Technical Report 239 (rerevised)
March 1989

UNIVERSITY OF
ROCHESTER
COMPUTER SCIENCE

The Rhet Programmer's Guide
(For Rhet Version 15.25)

Bradford W. Miller

The University of Rochester
Computer Science Department
Rochester, New York 14627

Technical Report 239 (rerevised)

March 1989

This work was supported in part by ONR research contract no. N00014-80-C-0197, in part by U.S. Army Engineering Topographic Laboratories research contract no. DACA76-85-C-0001, and in part by the Air Force Systems Command, Rome Air Development Center, Griffiss Air Force Base, New York 13441-5700, and the Air Force Office of Scientific Research, Bolling AFB, DC 20332 under Contract No. F30602-85-C-0008. (This contract supports the Northeast Artificial Intelligence Consortium (NAIC).)

SECURITY CLASSIFICATION OF THIS PAGE (*When Data Entered*)

REPORT DOCUMENTATION PAGE		READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER 239 (rerevised)	2. GOVT ACCESSION NO.	3. RECIPIENT'S CATALOG NUMBER
4. TITLE (*and Subtitle*) The Rhet Programmer's Guide (For Version 15.25)		5. TYPE OF REPORT & PERIOD COVERED Technical Report
		6. PERFORMING ORG. REPORT NUMBER
7. AUTHOR(s) Bradford W. Miller		8. CONTRACT OR GRANT NUMBER(s) N00014-80-C-0197 DACA76-85-C-0001
9. PERFORMING ORGANIZATION NAME AND ADDRESS Computer Science Department, 734 Computer Studies Bldg., University of Rochester, Rochester, NY 14627		10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS
11. CONTROLLING OFFICE NAME AND ADDRESS D. Adv. Res. Proj. Agency, 1400 Wilson Blvd., Arlington, VA 22209		12. REPORT DATE March 1989
		13. NUMBER OF PAGES 137
14. MONITORING AGENCY NAME & ADDRESS (*if different from Controlling Office*) Office of Naval Research, Information Systems, Arlington, VA 22217	US Army ETL, Fort Belvoir, VA 22060	15. SECURITY CLASS. (*of this report*) unclassified
		15a. DECLASSIFICATION DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT (*of this Report*)

Distribution of this document is unlimited.

17. DISTRIBUTION STATEMENT (*of the abstract entered in Block 20, if different from Report*)

18. SUPPLEMENTARY NOTES

none.

19. KEY WORDS (*Continue on reverse side if necessary and identify by block number*)

knowledge representation, hybrid reasoning, frames, logic programming

20. ABSTRACT (*Continue on reverse side if necessary and identify by block number*)

Rhetorical (Rhet) is a programming/knowledge representation system that offers a set of tools for building an automated reasoning system. It's emphasis is on flexibility of representation.

This document goes into almost excruciating detail about the internals of the Rhet system. It is intended to be used by those who will maintain it, those who wish to supply user-reasoning packages, and those who wish to add more builtin functions (unlike user Lisp functions, only builtins are capable of supporting side effects or backtracking).

20. ABSTRACT (Continued)

If you are strictly a Rhet user, this is probably not the manual for you. See (Allen and Miller, 1989), which covers the defined programmatic interface to Rhet. It is necessary that you understand the concepts explained therein before attempting to understand this document!

Abstract

Rhetorical (Rhet) is a programming / knowledge representation system that offers a set of tools for building an automated reasoning system. It's emphasis is on flexibility of representation.

This document goes into almost excruciating detail about the internals of the Rhet system. It is intended to be used by those who will maintain it, those who wish to supply user-reasoning packages, and those who wish to add more builtin functions (unlike user Lisp functions, only builtins are capable of supporting side effects or backtracking).

If you are strictly a Rhet user, this is probably not the manual for you. See [Allen and Miller, 1989], which covers the defined programmatic interface to Rhet. It is necessary that you understand the concepts explained therein before attempting to understand this document!

Contents

Chapter 1

Introduction

1.1 What is this thing?

The intention of this document is to give a solid overview of the code of the Rhetorical (*Rhet*) system. It is intended to be used by those who will maintain or extend the system. *It is not intended for users of the system*, nor is it a substitute for reading the code itself (which is, and should remain, reasonably well commented). This document is current for version 15.25 of Rhet, and should be updated periodically to reflect it's current state.

The philosophy of this document is: if you want to know what the code does, read the code. This document is intended to give a broad overview of the entire system's design to make reading the code possible. While most interface-level functions will be described herein, a large number of internal functions are not. The maintainer is advised to use a machine that will allow him to get to function definitions or present documentation strings, as this will greatly ease program understanding.

1.2 Acknowledgments

Special acknowledgment is given to Michael McInerny, who has invested a considerable amount of RA and programming time into the user interface, type subsystem, and user contexts; Stephane Guez for his work on structured types; Jun Tarui for his work on TMS and function typing; Nat Martin for his RA work on the parser; Steven Feist, for his RA work on the type subsystem; Jay Weber for his helpful comments and coding assistance; and everyone else who gave us suggestions on the design, those who suffered thru HORNE and are giving their experiences, and even those who built HORNE in the first place...

1.3 Help Us!

We want to know about problems you have using Rhet, inconsistancies with the manual, or suggestions on how any part of Rhet might be improved (*e.g.* to make it easier to use, the manual more clear, the index more functional, *etc.*). Send mail to one of the rhet discussion lists: Rhet@cs.rochester.edu or Bug-Rhet@cs.rochester.edu depending on whether you are asking a question, or proposing an enhancement (the former list is appropriate), or reporting a bug or inconsistancy. You may also contact the author directly using the address given on the title page of this document.

1.4 Overview

To begin, then, a quick overview:

Rhet is intended to be primarily a rewrite of the *HORNE* [Allen and Miller, 1986] reasoning system. The motivation for this rewrite is many-fold. Suffice it to say that the existing system had grown too crufty and fragile to be extended yet again. The intent of the rewrite is to:

- Provide an extended version of HORNE including extensions that had been planned to HORNE to support the current research. This involves:

1. Support (more or less transparently) of existing *HORNE code. This implies support for the* REP subsystem as well. This is not to be taken as a language definition, but a more general statement: functionality supported under HORNE should be upwards compatible with Rhet functionality[1].
2. Enhancement of the existing HORNE system to provide for contexts.
3. Enhancement to provide for reasoning about negation (rather than simply negation by failure).
4. Enhancement to provide some support for default reasoning.
5. Enhancement to provide some type of TMS.
6. Enhancement to provide a better user interface - both one that allows the user to debug Rhet code more easily, but also a more interactive form of the interface in keeping with Lisp Machine philosophy. (*E.g.* usage of ZMACS rather than a separate form editor).
7. Enhancement to support user defined *specialized* reasoners between specific types. For example, the TEMPOS system would be an example of a specialized reasoner.

- Provide a stable base for future enhancements to the reasoning system. This involves:

  1. A coherently designed system, rather than one that evolved over time[2]. The emphasis on the design should be maintainability and extensibility since future changes and enhancements *will* be made.
  2. A coherently implemented system, that is, one which utilizes a common, clear, well commented coding style[3].

[1]In general, we have often changed function names for no good reason other than we thought they needed changed. Porting Horne code should still not be difficult, just use a tag-table query replace on your horne code in the editor. We did take the time to post warnings and changes in the User's Manual.

[2]Of course, Rhet's design evolved over time too, but the interval was much smaller, and the designer was constant.

[3]Contrast to the HORNE system! Of course, this doesn't imply perfection, code can virtually always be improved. Mainly this is a side effect of sitting down and writing the system at one blow (more or less) rather than as a continuing stream of RA projects with functions being implemented and changed by different people every several months. Also not depending on low experience people to write the fundamental code helps too. Lessons obvious for anyone in industry.

- Provide a more efficient implementation of the existing system, building on our experience with the old system, and specializing the new system for the Common-Lisp language primarily, and the Lisp Machines secondarily[4].

1.5 System Compartmentalization

To support maintenance and extensibility, the Rhet system was designed as a compartmentalized system spanned several packages (in the common-Lisp sense). In theory, the control of imports and exports into packages should help define the interface between packages, and promote better high level design and hence more understandable code at the lower levels. Additionally, such is seen as a boon to debugging. Lower level code can be debugged independently of higher levels, so given a bottom-up implementation strategy, the higher levels can be built on the more stable bases of debugged[5] lower level code.

Note that the ZetaLisp **Defsystem** facility is taken advantage of, as are **Initializations**. More specifically, CL-USER::*Rhet-Initializations* is a list of forms to be evaluated (once) after loading Rhet. These are declared in the source via. **Add-Initialization** forms. See the "Porting" Appendix.

1.5.1 HNAME

The *Hierarchical Name Subsystem* is intended to be the basic KB for the Rhet system. It is the keeper of equalities and assertions relative to contexts. Additionally, it supports adding and deleting assertions, adding and deleting contexts, and adding equalities[6]

[4]We are, in general, willing to sacrifice some CL compatibility for better efficiency or ease of coding; we do not expect to move off the Lisp machine environment for some time, however, we wish to keep our options open for a port to, say, Butterfly-Common-Lisp should BBN ever finish it.

[5]At least partially.

[6]Equalities cannot be deleted *q.v.* section 12

1.5.2 UIC

The *User Interface Common Low Level* contains utility functions used throughout Rhet for interface to a limited extent with the user. *For example the dribbler is defined herein.*

1.5.3 TYPE

The *Type Database* is the keeper of type information, including function *typing*, and supports the retrieval of relationships between types. Additionally, it supports limited reasoning about type relationships.

1.5.4 HEQ

The *Hierarchical Equality Subsystem* provides a more robust interface to the equality maintenance facilities of the Hierarchical Name Subsystem. In addition, it provides limited functions to support equality based reasoning.

1.5.5 UI

The *User Interface Common High Level* contains utility functions shared by the various user interface packages. It includes the parser, for example.

1.5.6 TYPEA

The *Type Assertion Interface* takes type generation requests from the user, and installs them into the Type Database.

1.5.7 UNIFY

The *Unification Subsystem* implements the actual unification engine for Rhet. It supports typed rvariable and equalities for detecting if two horn clauses unify.

1.5.8 AXIOMS-F

The *Forward Chaining Axiom Database Subsystem* is the keeper of forward chaining axioms as defined by the user. It is responsible for adding and deleting axioms, as well as retrieving them as specified by the Reasoner or user.

1.5.9 AXIOMS-B

The *Backward Chaining Axiom Database Subsystem* is similar to the forward chaining one, but is kept separate both because of the different access methods needed, and because FC and BC are fundamentally differen, reasoning modes.

1.5.10 REASONER

The *Reasoner* subsystem implements the actual prover for Rhet. It accepts requests to prove a Horn Clause. and invokes the Unifier as needed, selecting axioms from the Backward Chaining Axiom Database, and executing Forward Chaining whenever new facts have been asserted. This is the actual 'Proof Engine' of the system.

1.5.11 QUERY

The *Query Interface* provides (separately) both a programmatic and user-enhanced interface for requesting proofs or doing unifications from Rhet.

1.5.12 ASSERT

The *Assertion Interface* provides fundamentally a user interface for adding axioms, equalities, or other forms of knowledge to Rhet. It is intended to be used either programmatically, or directly by the user via a enhanced interface.

1.5.13 RWIN

The *Window Interface* contains the functions associated with the high-level user interface, that are not contained in some other package (*e.g.* on the Symbolics, the rhet editing mode is in the **Zwei** package).

Chapter 2

Representations

This section documents the more important internal representations.

In order to provide a more object-oriented interface to the programmer, the representations are somewhat hierarchical. For example, all structures that are assertable include the structure **Rhet-Assertable-Thing**, which has the following slots:

Source-File-Name If the assertion was made from a file, it is recorded here, for user debugging purposes. **Rhet-Term-SFN** is the generalized accessor for this slot, regardless of the structure it is included in.

2.1 Is *That* a Fact?

Facts are the basic entities for asserted constants in the system (*e.g.* the user asserts that `[F A]`. Strictly speaking, the former is a fact, in the PROLOG sense, while the latter is an expression. The needs of the two entities overlap to such an extent that they share a supertye: Basic-Term. A Basic-Term has the following slots:

head which is typically the leftmost symbol in its horn clause representation, *e.g.* in `[F A B]` 'F' is the head. It is stored as an atom, though it is usually the printname that is important.

arglist which is a list of the function-term-accessors for each of the arguments to a fact or function term. In the above example, the accessors for 'A' and 'B' (in that order) would appear on this list. Canonical name accessors may also appear (*q.v.* section 15.3). Lisp objects may appear only if the Basic-Term is a Fact; Function Terms cannot have non-Rhet obejcts in their arglist[1].

type contains an **Itype-Struct** structure (*q.v.* section 2.5.1) which indicates the type of the term. It is assumed to be of the **T-U** type[2], which is the most general type, and the only type a Fact can take on. The **Itype**, **Dtype**, and **Utype** commands set this field for function-terms. It is ignored if the function-term has a canonical name, since the type can change depending on the context (via equality). Since it is of no real use, but the code does use it for historical reasons (the parser creates a fn-term structure with a type, and THEN it gives it a cname), it will go away in some currently unspecified future cleanup.

Note that Basic-Terms are a Rhet-Assertable-Thing.

The actual 'thing' normally passed between functions is a fact-accessor. This is an atom which is interned in a particular context, whose value is a **Fact** structure.

Fact structures inherit from Basic-Term and consist of the following additional elements:

truth-value carries the value of :**True** or :**False** (*e.g.* when `[NOT A]` has been asserted, the value for 'A' would be :**False**. The truth-value can also take on the values :**Unknown** and :**Unbound**; neither set except

[1]The reason for this is the main reason for having special Rhet objects in the first place instead of lists: we have to associate properties with terms that are not "atoms" in the lisp sense.

[2]Actually, the constant TYPE:*T-U-Itype-Struct* is used, the most general type.

internally to HNAME. :Unknown is a useful default, since fact structures are consed before they are asserted, and :unbound is used to *shadow out of a context* some *parent* fact with the same faccessor. That is, since contexts inherit, we track different truth values by using the same fact accessor in each context for a particular fact. We can then look for the fact accessor *in the closest context* (this context, then it's parent, then it's grandparent) and the first one found is the one we use. To assert a fact is true in some context and retract it in a child turns into putting an :**Unbound** instance in the child. That way, the parent, and contexts below the parent and above the child, still "sees" the asserted fact, while the child and it's children inherit the unbound one, and treat it as deleted.

index is a string representation of the index field of a horn clause, prepended with "INDEX-", thus for the horn clause `[[A ?x] <5 [B ?x]]` the index would be "INDEX-<5". The "INDEX-" prefix is used to distinguish it from the heads of facts, both end up being interned in the context the fact accessor is interned in, and their values are lists of fact accessor with that index or head, respectively. This is used to make lookup of facts by index or head (the typical case) faster than searching the entire context. In future, it is likely that the number of fields a fact is indexed on will grow. Using the context space to store these keys is a *matter of convenience*, it would be more general, but more wasteful of resources to cons up a separate hash table for each. Instead we lay contexts on top of the package system (purely to make debugging easier as the UI to packages is more straightforward than to hashtables) and rely on naming schemes to keep us from collisions. Fact accessors are all gensyms and guaranteed unique, fact heads are any atom, but extremely unlikely to overlap with the gensyms, and so long as no one goes and starts naming their facts beginning with "INDEX-¡" that should be safe too. It is likely that eventually a split will be made, since debugging contexts will be rare enough that not having the nice UI will be less of a handicap than these silly rules.

defaultp is set non-nil if the fact is to be considered *default* (*q.v.* section 15.1).

tag used by TMS.

support used by TMS, this is a list of support structures.

An example: Consider constructing the assertion [[F A B] <foo]. The user interface creates an accessor for each argument as a function-term, *e.g.* a gensym, setting it's value to a **FN-Term** structure. Then it does the same for the entire clause. For the above we would perhaps set X001 to be #S(:FN-TERM :HEAD A) and X002 to be #S(:FN-TERM :HEAD B)[3]. Then we set the accessor for the function term, perhaps F001 to be #S(:FACT :HEAD F :ARGLIST '(X001 X002) :TRUTH-VALUE :TRUE :INDEX "INDEX-<foo") plus any type information we might have in all cases, *etc.*

In addition, since facts are usually passed around via a Fact Accessor, there is a parallel structure for a **Faccessor** defined. It has the same slots as a fact (the symbol-value of a Fact Accessor is a Fact).

Assertable terms that can take on equalities are known internally as Function Terms, and are described by the **FN-Term** structure. Previous to release 15, a function term just shared structure with the facts, and were not distinguished internally except via special tests. Note that a special case of a function term, one with an empty arglist, is defined to be an **Atomic-FN-Term**. There is similarly an **Atomic-FNTaccessor** type defined, and corresponding predicates.

Function-terms have the following slots in addition to those they inherit from Basic-Term:

Canonical-Name is the canonical-name accessor for this fact (*q.v.* section 15.3), if it has one. Actually, this will be an alist of contexts and canonical names, because a fact may have different canonical names in different contexts.

Distinguished-Type This field takes a type similar to the type field, but is the type specified by the **Dtype** command. It is used solely for determining inequality.

Facts and Function Terms use the defstruct option to define their own printer. Therefore while the fact structure may be internally a (large) array, the fact whose head is A, and everything as default, will print as "[A]"[4].

[3]Of course, we would use a Make-FN-Term function call to actually create the FN-Term structure.

[4]Atomic function terms will not print the braces when they are present inside of other facts, function terms, or forms, for ease of reading. Thus one gets for the fact with two args [A B C] instead of [A [B] [C]].

2.2 A canonical by any other name...

Canonical names are in some sense the "real" names for objects that have them, and are used by the equality system. Unlike facts, software outside of the HNAME or HEQ packages should not be constructing or manipulating these structures directly.

The idea behind them is that two *different facts with the same canonical* name are equivalent in a weak non-logical sense. That is, given an *assertion* along the lines of [EQ [B A] [C]] then looking up the fact accessor for [C] and getting the canonical name, we would get the same canonical name as if we had done the operation on [B A].

There is software in the HEQ and HNAME packages for doing anything the higher layers of software may need on the canonical name structure (and they should be used), but for the record:

Canonical *name* accessors are usually what is passed between functions. Like fact accessors, these are atoms interned in a particular context, whose value is a canonical structure. This has the following slots:

set This is a list of the accessors for *all* facts in this union.

type This is the type of every item in the above set. Note that two facts of different types cannot be assigned to the same canonical name, unless one is a subtype of the other, in which case the fact that is of the supertype is considered to be specialized to the subtype.

primary This is a function term accessor to the FN-Term that is the preeminent member of the set. In the case of, for example having [EQ [MOTHER-OF TOM] [MOTHER-OF MARY]] and [EQ [MOTHER-OF TOM] SUSAN] SUSAN would be considered the primary. A primary must have an empty arglist. It must be of type **Atomic-FNTaccessor**. The primary is not semantically different from any other member of the set, but it is purely a UI issue: it is usually the "prettiest" member of the set to print.

references This is a list of fact accessors that this canonical class references. In the above example, the canonical class of TOM would reference the canonical class of SUSAN, since TOM is an argument of a function term that is in the class SUSAN, and thus if we were to assert that TOM was equal to something else, say THOMAS, we would want (indirectly) [EQ [MOTHER-OF THOMAS] SUSAN].

unequal This is a hash table used for testing inequality. Basically, if two facts are in the same context, then if they are unequal they will mention each other in their unequal fields. The exception is if they are unequal via Dtype rather than explicit assertion: see the description of inequality markers, below.

Type-Constructor-Function To handle REP style interactions more cleanly, if there is a constructor function associated with the class, mention it here. This can then be invoked, if necessary, to build the actual type. Many times we may be able to do without, e.g. simplifying [r-human-name [c-human alfred]] to [alfred] w/o constructing anything. We set this to a Form if it exists.

Inequality Markers This is used for speeding up inequality checks. Basically, since a large number of *different facts may be in a canonical class*, we collect the individual dtypes for the facts in this field. Then seeing if two canonical names are distinguished just involves doing an intersection on the two instances of this field, and if there is something in the resultant set, they are distinguished.

Accessor This is a pointer *to the Canonical Name Accessor for this structure.*

Value Cell Certain canonical names may have values, e.g. if we want to represent that [FRANK], an object of type T-PERSON has a R-BROTHERS role of the set {[BOB] [JIM]}, we cannot do equality between sets and function terms. Instead, we set the Value Cell of the function term [R-Brothers [FRANK]] to be {[BOB] [JIM]}. Alas, this is a hack, we are mixing the unification metaphor with assignment. It works until the term is involved in equality, then the result is undefined.

NM-Constraints Constraints that cannot be folded that have been added by **Define-Subtype** or **Define-Functional-Subtype** are kept on this list, if they are not monotonic. These are checked anytime the canoncial name is changed or updated.

M-Constraints Constraints that cannot be folded that have been added by **Define-Subtype** or **Define-Functional-Subtype** are kept on this list, if they are monotonic. These are checked anytime the canoncial name is changed or updated. Monotonic constraints (those that once proved satified will remain satisfied) are removed from this list as they are proved.

As for facts, Caccessors are also defined in parallel.

2.3 Do not fold spindle or mutilate...

While Facts are restricted to those horn clauses that do not contain rvariables, most horn clauses will. These sorts of clauses are defined by the Form structure in Rhet. Unlike facts, forms do not require accessors since they are never 'interned' in a context. Normally they are arguments to the Unifier, or to the Reasoner (as in (PROVE [B ?x]) or (UNIFY [?x (G . ?y)] [A (G C D)]). Forms are distinguished from lists, in that Forms are what make up clauses in axioms, and can therefore represent calls to builtins or lispfns. Only Forms may be unified against Facts. Lists are basically purely a Lisp type, though they can be used for dealing with matching several arguments within a form (as a rvariable), *i.e.* via the &rest syntax.

The main reason forms and lists are distinct[5] is that we must distinguish between [A B C] which is either the predicate A applied to arguments B and C or the *expression* [A B C] which is a unit for equality, vs. the Lisp list, (A B C). The difference is that in the former two, the arguments are essentially decomposable, but in the latter they are not. Where this makes a difference is the difference between [A (B C)] and [A [B C]]; the former must be a predicate (no equality possible), and has 1 argument, a list, while the latter may be an expression equal to [A D] if we know [B C] and [D] are equal, or even [F] if we know [A D] is equal to [F]. We can attach canonical names to fact structures, but not to lists. So we would run into a problem if [A . ?x] were legal (it isn't) and matched with [A [B C] [D]], would ?x be bound to [[B C] [D]]? If so, we have the problem that [[B C][D]] may inadvertently get treated as a unit, which it wasn't designed to be: we would have problems distinguishing between [A [B C] [D]] and [A [[B C] [D]]]; in fact the semantics

[5] *If you don't immediately see the reasoning behind all this, I can't blame you: during the first year of the project* I didn't think we needed to distinguish between the two!

of [A] vs. [[A]] vs. [[[A]]] become unclear. Further, leaving everything in lisp list syntax gives problems with things like (?x . ?y*(foo bar)) since that looks like something illegal to Lisp. Distinguishing them makes certain things more clumsy (i.e. vararg type predicates), but increases expressivity.

Forms have the following slots:

value the elements of the form, as a list. *E.g.* from the horn clause [?x B ?y] it would be (?x B ?y).

rvariables A list of the rvariable structures used in the form, directly or in subforms.

truth-value As for a Fact, in a goal or clause in an axiom, distinguishes between, say, [A B] and [NOT A B]

type an **ltype-Struct** structure for the type a form returns, if it is known. A ***T-U-ltype-Struct*** value (the default) means it is not known, and so most general.

Where-Posted If this form is a constraint, then this is the continuation point at which it was posted. This is used to GC constraints off of variables when we want to unbind them. This is likely to change in version 16, because a global proof tree will have this information. Either this slot will be a pointer to the tree, or it will disappear entirely. I won't know for a month or so yet.

Forms are probably the most useful of the various structures, as they will be the primary datum being thrown around above the lowest levels. One definition I will add here: The complexity of the form is defined as the lowest level a rvariable is present on. Thus a form with no rvariables is equivalent to a fact in some sense, though it may not be added to HNAME's KB. A form whose deepest rvariable is on level 1 is called 1-complex, which is what several functions expect at worst case as arguments. Thus if you, as a middle or high level routine have a 3-complex form you want to simplify, and the simplification function only accepts 1-complex forms, you need to extract the deepest part of the horn clause that contains a rvariable (which will itself be 1-complex) and turn that into a form and hand that to the function. If it simplifies to something without a rvariable, you can use that as a replacement for the form you extracted and continue[6]. Better

[6]Actually it is highly unlikely that a form with a rvariable would simplify in this sense, but take this as illustrative.

coding practice would define a recursive version of the function that takes n-complex forms but this may not be practical in some cases. (*E.g.* where anything larger than a 1-complex form is guaranteed to fail).

2.4 Rvariable: Rochester's Weather

One of the entities that can exist on a form is a *rvariable*. What we represent in a horn clause as ?x has a much more complex internal representation. It is (again) a structure[7], and does not need an accessor since, like forms, it is not something we add to the database. Since we have to distinguish between the printed version of a rvariable, *e.g.* the ?x we use in a query and an axiom we have in our database that uses a rvariable with the same printed representation.

Here are the slots a rvariable has:

pretty-name what we see when we print the rvariable, *e.g.* ?x. This does not include what the user interface might want to print out in terms of the type of the rvariable, or the constraints.

actual a gensym that distinguishes this rvariable from any other, and makes it easy to see that while debugging[8]. (EQ vs. EQUAL)

type the type a rvariable has, if it is typed. By default is is of type *T-U-Itype-Struct* which is a constant for the universal type T-U. A rvariable may only have a constant assigned to it that is a subtype of or equal to the rvariable's type.

Binding What the rvariable is bound to, if it is.

[7]Structures are considered efficient ways to represent things in CL and are implementation dependent representations to keep them that way. That is, on the Symbolics^{TM}, it is stored as an vector, *but need not be on a* SUN^{TM} *using* Lucid^{TM} CL.

[8]If *HDebug* is non-nil, variables print out annotated with a portion of the printname of this symbol.

Culprit-Continuation The car of this list is the continuation to invoke if this var is culprit. This is likely to go away in release 16 in favor of a indirect entry through the proof-tree.

Where-Bound What part of the stack we must unwind to to undo binding. This is likely to become a pointer to the proof-tree we were bound at in release 16 since it improves efficiency.

constraints for the post constraint mechanism, these are the constraints placed on a rvariable. Right now, the constraints themselves have slots indicating the continuation level they were POSTed on. This is likely to change in version 16.

2.5 It's Not My Type

2.5.1 The Itype-Struct

The **Itype-Struct** structure is used in most of the above structures to describe the type of the object. Normally it would not be manipulated outside of the TYPEA or TYPE packages. Currently it consists of the following:

intersect-types A list of all simple types that intersect to make this type.

minus-types A list for subtracting from the above result using a calculus of types. (*E.g.* the type `[foo - bar]` would have type `foo` as a intersect-type and `bar` as a minus-type).

fixed-flag Thought to be useful if set that the type is the immediate type of the element, and cannot be further constrained. Thus if I said that **Males** and **Females** are an exclusive partition of **Humans**, (thus anything that is a **Human** must be either **Male** or **Female**) this would let me create an element that is a **Human** that could not be specialized by the system into either **Male** or **Female**[9].

[9]This isn't actually used except by the Itype declaration, and currently isn't used for anything; it may turn out to be useful, but it hasn't been investigated. One possibility is if you create prototypical things and you don't want the system making these prototypes equal (*i.e.* using equality) to some 'actual' thing. Instead you would get an error.

Itype-structures exist because simple types are precomputed, and we need some way to represent types that are not in the table, In particular, unnamed intersections, and eventually, equations in the type calculus. Thus, for generality, all structures that have the type marked use an Itype-structure, which the type subsystem can figure out how to deal with in terms of accessing the type table.

2.5.2 The REP-Struct

The REP-Struct is used to maintain the role, constraint, and other definitions pertaining to structured type objects. It has the following slots:

Roles A list of the roles defined on this type. It does not include the inherited roles from parent types, but only the local additions and redefinitions.

F-NM-Constraints This and the following constraint slots hold a list of things to be asserted or proved at various times. This list contains foldable non-monotonic constraints, *i.e.* those for which something can be asserted at instance creation time to make the constraint hold, but must be tested after manipulation of an object, *e.g.* after processing an equality addition.

F-M-Constraints These constraints are foldable and monotonic, that is, they can be asserted at instance creation time, but never need to be tested for. Equality is an example, since equality cannot be retracted.

Initializations A special case of F-M-Constraints that appear as a list of assertions to be made, rather than relations that must hold.

NF-NM-Constraints Non-Foldable Non-Monotonic constraints must be proved rather than asserted, and may change during the proof.

NF-M-Constraints Non-Foldable Monotonic constraints must also be proved, but once proved will not change so need not be reproved.

Functional-Roles A subset of the Roles slot, these roles will have functions associated with them predeclared.

Type The typename itself for this structure. A symbol, usually in the Type-KB package; it's value would then be an **Rtype** structure.

Relations-Alist An alist of relation names and the list of the relaions.

2.6 Truth

There are *axiom* structures for both forward chaining and backward chaining axioms. These are, for the most part, identical, and the **:Include** option on defstruct makes the **Basic-Axiom** structure shared between the two. The **BC-Axiom** structure is just an alias for this, adding two fields:

Cache BC axioms (will) cache their proofs: car is context, cdr is list of proof-cache structures.

References For (future) cache flushing.

while the **FC-Axiom** adds a field: **Trigger** which is a **Form**.

Without further ado, the **Basic-Axiom** structure:

LHS A **Form** that represents the left hand side of a horn clause, that is, the conclusion.

RHS A list of **forms** that are the prerequisites for concluding the LHS.

index An ascii string, the index field of the axiom.

context The context this axiom is asserted to. It is considered valid in all subcontexts[10].

[10]Unlike facts, deleting or modifying axioms in a subcontext is NOT supported.

defaultp If *non-nil, the axiom* is DEFAULT. It will *only* be used by the reasoner when default reasoning is enabled.

Global-Vars The rvariables used in this axiom that are global (not local).

Local-Vars The rvariables used in this axiom that are local (*i.e.* in LHS if BC, Trigger if FC, unless inside of a structure (may still be global)) typically this is determined at call-time, for BC axioms, and at parse/compile time for FC.

Key For indexing (hashing), this is the computed hash-key for this axiom.

Axioms are distinguished from facts in that they can contain assertions with rvariables in them, and use any form to specify an axiom. Thus lisp-lists, lisp atoms, builtins, lisp-functions are all legal constituents, except the predicate position must be a builtin, lisp-function or predicate.

2.6.1 Rhet-Set

This is the internal representaiton for sets in Rhet.

Currently defined, but unused (in version 16).

Set A list of objects in the set.

Type Either :ORDERED or :UNORDERED (the default), this indicates if the order of objects in the Set slot is important.

Cardinality NIL, if the cardinality of the set is unknown or not fixed. An integer *if the cardinality is* both known and fixed. Thus, if there are three elements in the Set slot, if Cardinality is NIL, then we don't know how many members are actually in the set (though there must be at least three). If Cardinality is three, then the set is completely known and fixed, and if the cardinality is greater than three, then we know how many members of the set there are (and it is fixed) but we only know three elements so far. Obviously a Cardinality less than the current number of elements in the Set slot is an error.

2.7 Minor Structures

2.7.1 Ref-Record

This defstruct will be used for intelligent cache flushing (unused as of version 16). It consists of the following slots:

References Other axioms that must be flushed if we are.

Siblings Other axioms like me.

2.7.2 Undo

The **Undo** structure is used to help undo equalities that are added that cause illegal indirect unions. It records the canonical name of the generated (new) class, as well as the canonical names of both the old classes that were unioned together. Additionally, a 'timestamp' (actually an integer count) is provided as a quick sanity check on the linked list the Undo structure is part of.

2.7.3 Hier-Context

This structure is used inside of a context to indicate the parent context as well as a list of the child contexts for this context. It is the value of the WHO-AM-I constant in a context (*q.v.* 15.1).

2.7.4 Rtype

This structure is used within the TYPE system. Type "names", *e.g.* the contents of the plus-types slot in an **Itype-Struct** structure are atoms that are set to an **Rtype** structure. It consists of the following:

supersets the defined (immediate) superset of the type, *e.g.* when I create a type, I must state what it is a subtype of. This is assumed to be an immediate supertype.

subsets A list of all defined immediate subsets of the type.

index the array index into the type table for this type.

partitioners A list of the partitioners of this type.

partitionee If this type partitions some other type, this is that type.

Once again, the above is described for completeness, only the TYPE and TYPEA packages are ever expected to manipulate these structures.

2.7.5 Proof-Cache-Entry

The Proof-Cache-Entry structure is defined for axiom structures. This will allow us to (eventually) implement a goal cache. It consists of the following:

Key The calculated Key of the result, for a possibly quicker match

Proved What it is that we proved.

How-Proved The justification / proof tree for this item.

The idea is that since axioms (compiled or uncompiled) end up being closures, there is some need for these closures to share state with each other. That is, as proofs of an axiom are completed, you still want other instances to know about the completion, even if they are currently inactive. So, we put it on the axiom structure, which a closure is already associated with (one closure implements one axiom), and it can keep checking "itself" to see if new proofs have been done. (In fact we use a fun little pointer game; see the code.)

2.7.6 Defined Types

The following Deftypes are supported:

Atomic-FNTaccessor A function term accessor for a function term that Satisfies Atomic-FNTaccessor-P, that is, it's related function termis of type Atomic-FN-Term.

Atomic-FN-Term A function term that Satisfies Atomic-FN-Term-P, that is, it is a function term with a null arglist.

Arbform An arbitrary form. This is Deftyped to (OR FORM KEYWORD LIST ITYPE-STRUCT RVARIABLE). That is, the parameter of this type is allowed to be a rvariable, a fact (fact-accessor), a form, a keyword (the way Lisp atoms can be used *during* unification) or (in the case of arglist processing, or more generally) a list[11]. This is the most general type of object in Rhet for things the user can have in a horn-clause. Note that an Axiom (which is itself a type) is NOT an Arbform. Additionally to support the Type? builtin, an Arbform may be an Itype-Struct structure.

Culprit-Type The type of a Culprit, as will be passed to a generator upon needing an additional value. The name is taken from the intelligent backtracking literature...

Context-Type The type of a CONTEXT parameter. It's actually a package, but use of this deftype will keep such lower level details out of upper level code.

Frozen-Axiom The type of a frozen axiom. Axioms may be frozen (internally) via functions such as Freeze-Axiom and 'thawed' via functions like Thaw-Axiom.

Frozen-Form-Part The type of a frozen Form. See Frozen-Axiom, above.

[11]The astute reader will know that FORM or RVARIABLE, being a structure, is also an ATOM on the Symbolics and Explorer. The definition is to be taken more literally — as documentation

Generator The *type of a generator. Generators are* what builtins (among other things) return. They are basically closures. The idea is that one is associated with a subproof, and when called return the next subproof.

Hash-Index What we use to look up facts fast, that is the type of the KB index, not the printed index of a fact.

Legal-Goal The type of an object that Rhet considers to be legal as a goal, as *for backward chaining*.

2.8 Handling Errors

Errors in the Symbolics and Explorer systems are built on Flavors. The basic idea is that a condition is signalled which will invoke methods on a *flavor appropriately. Depending on the flavor method* for the particular error, additional information or a documentation string is provided. The signal, then may be caught by an enclosing form, and possibly handled, *e.g.* if it is proceedable, or possibly the debugger will be expected to deal with it. The base flavor for Rhet errors is **Hname:Rhet-Condition**. *Other* subflavors (errors) defined on top of this base are:

Heq:Rhet-Equality-Problem This is used to signal a problem with recursive equalities. It supplies two proceed options: :undo, which will cause the offending equality to be undone and tossed, and :continue, which will cause the equality to be added anyway, generating an inconsistency.

As error handling is somewhat more machine dependent at this time (since a CL standard has not been defined at this date), more details should be garnered from examining the actual code involved.

Chapter 3

The User Interface

The user interface to Rhet is described in a separate document: **The Rhetorical Users Manual** [Allen and Miller, 1989]. What follows is a description of some UI internals.

3.1 Programmatic User Interface

The main thing to remeber is that when you build a system to use Rhet as your KR engine, you should have your system's pacakge USE the packages Asswert, Query, and UI (at least). That way, your system has access to Rhet's exported functions that are intended to be used and unlikely to change without using a package prefix. This can make maintainance easier, as between Rhet releases if something breaks, you may be able to quickly localize the problem to a call you make to some "internal" Rhet function that we didn't promise to keep constant (and hence didn't warn you would change).

3.1.1 Assert

Documentation for the Assertion package:

Rhet-Dribble-Start *File-Spec* &Optional (**Mode** :**Both**)
Enable recording of Rhet interactions into the file. The default mode is :**Both**, that is both queries and assertions will be recorded, otherwise use :**Query** or :**Assert** to be selective. In either case, the functions are represented in the file, along with their results commented out. You can change the mode while dribbling by supplying a **Nil** File-Spec. The File-Spec may also be a stream. In this case, the function will not attempt to open the stream, just use it.

Rhet-Dribble-End *NIL*
Disable recording of Rhet interactions.

Builtin-Trigger-Exception-List This is a list of the BuiltIns that *are* valid trigger candidates. All others are rejected.

3.1.2 Query

Documentation for the Query package. Most take an optional keyword :**Context** argument.

Find-Form *A-Formula*
Will take a form and try to find all the Facts that unify with it. This function is a good example of how to call a builtin when not inside the prover.

Find-Form-With-Bindings *Formula*
Will take a Form and try to find all the facts that unify with it, and return the bindings.

Chapter 4

Tutorial — Lisp Predicates

4.1 When to use a Lispfn

Lispfns, that is lisp functions that are to be called from Rhet and act as predicates or assertions, allow for a very simple and convenient way to interface Rhet to some lisp system. This system may be:

- a primitive backend, acting as an efficient representation mechanism for a particular kind of knowledge that Rhet will want to manipulate, *e.g.* time intervals, parsed sentences, ...;
- a way to imporve performance when a predicate can fold naturally and easily into Lisp code that does *not require* the power of a builtin.

Lispfns, however, are much more limited than builtins. Their advantage is that they are much easier to write and debug than builtins, and require no deep understanding of the Rhet system as builtins do.

These limitations are:

- Lispfns cannot backtrack, builtins can. That is, a lispfn may only succeed or fail, and must be deterministic. They cannot bind any variable[1], because that can only be done in the context of backtrack handling: one provides code that handles the backtrack and tacks it onto the variable so should the variable need to be unbound this code can be called. Since lispfns do not provide this function, they cannot bind a variable.

- Lispfns should not have side effects, and cannot have undoable side effects. When Rhet does a proof, it may call a lispfn one or more times in the process of doing a proof. There is no guarentee of which or any of these calls will *end up being logically responsible* for the final proof(s) returned to the user. Thus if the lispfn has side effects it may do things that are *unrelated to the actual thing proven*, they will only be related to the means by which they were proven. For example:

```
(4.1)     [[P ?x] < [Q ?x ?y]
                    [My-Lispfn ?y]
                    [R ?y]]
```

It is possible in 4.1 to have My-Lispfn invoked on each value of ?y for which [Q ?x ?y] is true, though only one value of ?y allows [R ?y] to be provable. This last one is the only effective one used to prove [P ?x], but it is not distinguishable by My-Lispfn from the other proofs. They may have been proofs from different parts of the proof tree, or multiple-proofs due to a Prove-All. Further, the rule that tried to prove (directly or indirectly) [P ?x] may fail, and no notification is given to functions called from parts of the proof tree that get lopped off.

[1]Partially because this would potentially lead to backtracking, but really because the variable binding mechanisms associate with bound variables certain information that is not provided by the simple lispfn mechanism and requires the much more complex macros used by the builtins. It is, in fact, relatively straightforward to write a builtin with the limited functionality of a lispfn, but that can bind variables, however since this involves a more careful understanding of the stack and how Rhet passes variables and does proofs lispfns have been limited to not take variables. Invoking a lispfn with a variable unbound will invoke the debugger.

4.2 Writing Lispfns

Write a lispfn just like you were writing a lisp predicate. The easiest thing to use would be the **DefRhetPred** form, which is documented in [Allen and Miller, 1989]. Since you are making the assumption that all arguments to the lispfn will be bound when you are called, most predicates are quite straightforward. For example, suppose we wish to add date-string recognition to Rhet. Rhet already knows about strings to a limited extent, so we might just do:

```
(4.2)   (DefRhetPred Date-String? (input-string)
            "Takes a string and returns non-nil if it is a date"
            (time:parse input-string))
```

This works to the extent that input-string always represents a legal time, since **Time:Parse** will drop into the debugger otherwise. I leave it as an exercise to the reader to write a version of **Date-String?** that would actually return **NIL** if the input string were not a legal time.

4.3 Language Constructs

These definitions also appears in the User's Manual:

Declare-Lispfn *<Name> Query-Function-Symbol* &Optional *Assert-Function-Symbol Type-Declarations*
Declares to Rhet that Name is not a predicate but a Lisp function; Rhet will recognize those <Name>s as calls to Lisp functions. If the Reasoner is attempting to prove an axiom that has been declared a Lisp function, it will call the Query-Function-Symbol (passed as a symbol to allow it to be incrementally recompiled). If it attempting to add a predicate to the KB whose head is declared with an associated Assert-Function, it will call the Assert-Function-Symbol rather than add it[2]. Lisp Query-Functions

[2]It will still forward chain as if it had added it, although it may not detect a loop!

should only return "t" or "nil" which will be interpreted as true and false respectively. The optional Type-Declarations are a list of symbols representing the types of the arguments expected by the lispfn. This will be used for runtime debugging interaction, and as a hint to the compiler. Declare-Lispfn will check in the KB to see if Name has previously been used in constructing Rhet predicates, and if so, warn the user about potential problems[3].

DefRhetPred *Predicate-Name* (**Argument-Lambda-List**) &Body *Body*
New Defines a Rhet predicate Predicate-Name that takes arguments according to the Argument-Lambda-List. Body is either a series of indicies and RHS definitions, or lisp code. These may not be mixed. Only the lispfn usage of DefRhetPred is described here.

The Argument-Lambda-List may be made up of the following:

1. & keywords, specifically:

   &Any the following variables (until the next & keyword) may be bound or unbound when the predicate is invoked, or may be bound to terms that are not fully ground. This is the default when no & keyword appears.

   &Bound the following variables are guarenteed by the programmer to be bound when the predicate is invoked. It is an error to attempt to prove the predicate without passing a fully ground term in this position. If (**Declare** (**Optimize Safety**)) appears, erroneous usage will signal an error[4]. If forms follow this keyword, variables embedded in the forms must be bound.

   Currently unsupported with lispfns.

   &Forward this must be the last &Keyword specified. If present, the **DefRhetPred** defines FC axiom(s) rather than BC. The form(s) following this key are the trigger(s). If none are present, it is the same as having specified **:All** as the trigger. No unbound variables may appear in the

[3] A form that has been read by the parser before this declaration will have a head that is a Fact structure, and after will have a head that is the Name symbol. This distinction is picked up by the prover, so the former will never be recognized as a Lispfn at proof time, and the latter never looked up in the KB.

[4] Future functionality

trigger. The other part of the lambda list becomes a pattern for the fact to be asserted if the body of the **DefRhetPred** is proved[5].

&Unbound the following variables are guarenteed by the programmer to be unbound when the predicate is invoked. If forms follow the keyword, variables within the forms must be unbound. It is an error to attempt to prove the predicate with any of the variables bound. If **(Declare (Optimize Safety))** appears, erroneous usage will signal an error[6].

&Rest the following variable must be of type T-List (it will be changed if it is not), and will be bound to all the remaining arguments of the predicate when attempting to prove it. This is just like the normal usage of &Rest in forms.

2. Rhet variables, which have the properties of the closest preceeding & keyword, as above.
3. Rhet forms, which are pattern matched against the form being proved to see if this predicate is applicable.

The body then consists of the following:

1. If the first object is a string, it is considered a documentation string for the predicate.
2. If the first object (after the optional documentation string, if any) is a **Declare** form, it is taken to be declarations about the predicate, as for CL. All CL declarations are legal for lispfn predicates. Other declarations that are legal include Foldable, Non-Foldable, Monotonic, and Non-Monotonic.
3. After the optional declarations, if the next object is not an index, the remainder to the body is taken to be an implicit **Progn** that defines an anonymous lisp function that will act as the predicate. At run-time, the lisp function will be evaluated, and if it returns non-nil, the predicate succeeds. Note that only one **DefRhetPred** may be defined on a predicate if it is to expand into a lisp function. Note that if the lisp function will have embedded Rhet forms that contain references to the variables

[5] Currently, it is an error to have both &forward, and the body containing a lisp function.

[6] Future functionality

in the Argument-Lambda-List, the **DefRhetPred** must be surrounded with the #[and #] special characters to put these references into a single environment, to assure they all refer to the "same" variable.

Chapter 5

Tutorial — Builtins

5.1 When to use a Builtin

If a lispfn isn't sufficient, either because you want to handle variables in your arglist, or want to declare side-effects, you need to use a builtin. The interface of a builtin to the reasoner is substantially different from a lispfn. A *lispfn takes the arguments* it expects to have as if the rhet form were a lisp function application, and returns a non-nil result to indicate success. A builtin takes two additional arguments before the other arguments it would expect from the form: a failure continuation and a continuation for variables that were protected before calling the builtin (more on protection in a second). A builtin can do one of two things. It can invoke the failure continuation if there is no possible solution, or it can returns a generator which will return solutions each *time it is called until there are no solutions left*, and then it will invoke the failure continuation. This generator is expected to be a function of one argument. The argument is the so-called culprit. If it is NIL, the next proof should simply be supplied. Non-nil culprits will be covered under the *Writing More Advanced Builtins* section below. It is possible, however, to always ignore the culprit and merely generate the next possible proof. The culprit is used for the failure-driven backtracking mechanism which will be described in more detail below.

Generators are lexical closures in common lisp. Normally the failure continuation, for example, are passed via the lexical environment to the closure. Remember the closure will only be invoked with one argument so it is important to have everything else you may need in the generator's lexical environment.

but it could be better.

Several macros have been written to make writing a builtin more straightforward.

5.2 Writing Simple Builtins

Lets start by looking at two simple builtins. The first one will be completely deterministic. This is still more powerful than a lispfn, because it can still bind a variable. But being deterministic, it must either succeed or fail: it will not backtrack.

```
(5.1)   (ADD-INITIALIZATION "Define builtin:  HFUNCTION-VALUE"
            '(DEFINE-BUILTIN 'HFUNCTION-VALUE "FUNCTION-VALUE"
                             '((OR FORM FNTACCESSOR) T) :NF-NM)
            () 'USER::*RHET-REPEATABLE-INITIALIZATIONS*)

(5.2)   (DEFUN HFUNCTION-VALUE (FAILURE REINVOKE TERM VALUE)
            "Succeed if the value of the function term Term is (unifies with) Value"
            (DECLARE (TYPE CONTINUATION FAILURE REINVOKE)
                     (TYPE ARBFORM TERM)
                     (TYPE T VALUE)
                     (OPTIMIZE SAFETY))

            ;; this is strictly a lookup function and is side-effect free.  Since the form
;; can have at most one value, it is also deterministic.

(RATIONALIZE-ARGUMENT TERM :DO-FORMS T)
(RATIONALIZE-ARGUMENT VALUE :DO-FORMS T)

;; react somewhat differently depending on where the rvariable is
;; (or if there is one).

(TYPECASE VALUE
   (RVARIABLE
    (UNIFY-VAR FAILURE REINVOKE VALUE (FUNCTION-VALUE TERM)))
   (OTHERWISE
    (BUILTIN-EXECUTES-ONCE-ONLY FAILURE REINVOKE T
        (COMPLEX-UNIFY (FUNCTION-VALUE TERM) VALUE)))))
```

Note the usage of the two parameters before the form that is used to call the builtin; these are standard for all builtins. We first use the macros **Rationalize-Argument** to handle the arguments to the builtin that will come from the Rhet form being proved. This macro makes sure that when we reference one of our arguments, we get what the variable was bound to and not the variable itself if it is bound.

Here is another builtin that is nondeterministic (it can backtrack):

```
(5.3)    (ADD-INITIALIZATION "Define builtin:  HGENVALUE"
             '(DEFINE-BUILTIN 'HGENVALUE "GENVALUE"
                               '((OR RVARIABLE LIST) T-LIST) :NF-NM)
             () 'USER::*RHET-REPEATABLE-INITIALIZATIONS*)
```

```
(5.4)   (DEFUN HGENVALUE (FAILURE IGNORE RVARIABLE* LISP-EXPRESSION)
          "Sets the Rhetoric rvariables to the first value in the lists returned by
          evaluating the lisp-expression.  Other values are used for backtracking.
          This is like the current definition, but if the lisp expression returns
          multiple values, it will bind them to each rvariable in turn, the car of each
          value returned.  Additional values beyond the number of rvariables supplied
          are ignored.  Additional cdrs of the lists returned are used for backtracking.
          Should one list be shorter than another, it will supply a value of NIL if
          backtracking proceeds to that point."
          (DECLARE (TYPE LIST RVARIABLE* LISP-EXPRESSION)
                   (TYPE CONTINUATION FAILURE)
                   (OPTIMIZE SAFETY))

          ;; This functions lisp-expression argument will be a list, not a real function
          ;; (otherwise fiddling with it's arglist for those terms that are bound
          ;; rvariables would be hard!)  At this point, the vars should be bound, or the
          ;; lisp function may get a surprise!  (sorry, not my problem!!!)

          (IF (NOT (CONSP RVARIABLE*))
              (SETQ RVARIABLE* (LIST RVARIABLE*)))   ;make it a list.

          (ASSERT (EVERY #'RVARIABLE-P RVARIABLE*) ()
                  "Not all the forms passed to GENVALUE
                  in the Rvariables position are rvariables")

          (LET ((RESULTS (MULTIPLE-VALUE-LIST (EVAL LISP-EXPRESSION))))
             #'(LAMBDA (CULPRIT)
                   (DECLARE (TYPE CULPRIT-TYPE CULPRIT))
(IGNORE CULPRIT) ;not used.  Could be, though.
(LET ((FAIL T))
   (COND
      ((EVERY #'NULL RESULTS)
       (INVOKE-CONTINUATION FAILURE)) ;done.
      (T
      ;; generate the next proof of the goal.  The interpreter
      ;; doesn't cache (yet, anyway) to make debugging easier.
       (WHILE FAIL
          (SETQ FAIL NIL)
          (SETQ RESULTS
                (MAPCAR
                   #'(LAMBDA (X Y)
                         (CHECK-TYPE Y LIST)
                         (UNLESS (UNIFY-RVARIABLE X (CAR Y))
                            ;; this one fails,
                            ;; finish taking cdrs, then try next
                            (SETQ FAIL T))
                         (CDR Y))
                   RVARIABLE*
                   RESULTS)))
       :GENVALUE-RESULT)))))) ; have to have some ''Justification''
```

The important thing to note here is that the result of the builtin is a CLOSURE. This closure is designed to be called and return a new binding (as a side effect) of the variable passed to the builtin. It does this with the **Unify-Rvariable** function. It returns a justification structure, which in this case is just a keyword[1].

[1]Normally one would return a list of the fact or axiom structures needed to cause the builtin to succeed, but in this case since

It is importatn that when the generator exits it pass back some non-NIL result. The reasoner has no way of noticing the side effect of destructive binding of some variable. A NIL justification, then, is interpreted as a failure.

5.3 Writing More Advanced Builtins

The main things this section will discuss are

- What a Builtin may have to know about how the interpreter and compiler works (more advanced control strategies).
- How to identify a culprit.
- How to handle a culprit identified elsewhere.

If you find the above necessary for the builtin you wish to construct let me add one cautionary note: there is no substitute for READING THE SOURCE CODE. Find a builtin that already does pretty much what you want, and understand it thouroughly.

<<<<— [More text to be added here!]

5.4 Language Constructs

Define-Builtin *Symbol User-Name Argument-Type-List Builtin-Type* &Optional *Assert-FN-Symbol Undo-Compiled-Function*
This function defines Symbol to be a builtin, and optionally supplies a function to call for undoing.

we are dependant on an ephemeral call to some arbitrary lisp function, we can't do so.

The function, like all builtins, is expected to return a generator. Not supplying an Undo function to Define-Builtin is defining the function as side-effect free. Builtins with side effects that cannot be undone are encouraged to supply a Undo function that sends a warning to the user. The Argument-Type-List is a user-interface specified list of the types of the arguments the builtin expects, as a hint to the parser, and to allow the high level user interface to catch syntax errors. The Builtin-Type is a keyword which indicates to Rhet the kind of builtin it is, for compilation or constraint purposes, *e.g.* foldable, non-monotonic. The User-Name is how the builtin will be known to the user and should be a string. The optional Assert-FN-Symbol should be the symbol for a function to be called if this builtin is asserted, as in appearing in the LHS of an FC axiom, or as an argument to **Assert-Axioms**.

Rationalize-Argument *Thing* &Key *Do-Forms* Macro

Call this on args to builtins to change the vars to be their bindings, and, if Do-Forms is non-nil, forms to be simplified. Otherwise the argument is left alone.

Real-Reference *Thing*

If thing is a variable, return it's binding, otherwise Thing itself.

Calling any of the below functions will return a closure which will do the indicated operation. All functions take a non-optional failure-continuation parameter for passing the continuation to be taken if the function cannot succeed (and cannot identify a culprit). This closure is then normally called immediately to attempt the indicated function. Successive calls will generate additional possibilities as appropriate. This closure is returned as the second value from a builtin: some builtins without backtracking possibilities will simply return their computed value and a second value of nil to save the work of Consing up a closure.

Note that parameters may be preprocessed by the interpreter, but rvariables should not be. This is because the intelligent backtracking capabilities rely on knowing where a rvariable was bound if a bound rvariable cannot be unified with anything to prove the subgoal. Each of the following functions exist in two forms. For a function Frotz, calling Frotz does the indicated operation. Normally only the interpreter will make such calls. Calling Frotz-mac (the macro form) will expand into Lisp code that can be compiled by the axiom compiler.

In the documentation below, the failure continuation parameter, which is always the first parameter, and the second value returning the closure are omitted.

First, here is a list of the macros this library defines which are meant to be used to appropriately manipulate the closures, failure continuations, and what they return. For a good example of the calling sequence, look at the axiom interpreter functions, *i.e.* Interpret-FC-Axiom and Interpret-BC-Axiom. The former is simpler, and should be examined first.

VERSION 16 NOTE: The main change Version 16 should have is to change the way global variables are handled. The following macros will no longer "protect" a variable, rather, the variable will point to a global proof tree/table indicating where the variable was bound, *etc.*. This means that

1. Variables will no longer need to be scanned at each proof level which should be a substantial performance improvement, particular with regard to Rhet-consed lists (*i.e.*lists whose car is a rhet object and whose cdr is a rhet variable which is bound to a list cell).
2. [Cut] will be able to be fixed (at long last).

Define-Continuation *Continuation-Name* &Body *Code-That-Can-Invoke-Continuation*
Sets up a continuation at this point. Only code within the body of the Define-Continuation may invoke it.

Invoke-Continuation *Continuation* &Optional *Culprit*
Invokes a continuation. Anything on the stack above the continuation point is lost. Note that continuations are expected to accept a Culprit argument[2].

Debug-Continuation *Continuation* &Optional *(Msg "Establishing Continuation of ") Pro*
This is used in the other continuation macros to simplify debugging. Pro contains the protected rvariables, if there are any.

[2] Actually, this is a function, to aid in debugging; eventually it should be declared inline.

Create-Generator *GenVar Creator*
Uses Creator to create a generator, assigning it to Genvar.

Invoke-Axiom-With-Failure-Continuation *Axiom* &Rest *Arglist*
Invokes the Axiom on the passed Arglist but adds an additional (first) argument: a continuation point that on failure will return to the code following this macro.

Invoke-Generator *Generator* &Optional *Culprit*
Builtins, *etc.* return generators (closures) which need to be invoked to generate "proofs". This macro does the work. It passes a Culprit, which can be NIL if no culprit is known[3].

Invoke-Generator-With-Failure-Code *(Generator Culprit)* &Body *Failure-Code*
This function is like Invoke-Generator except that if the Generator fails, the Failure-Code is run before the macro returns.

Invoke-Deterministic-Builtin *Builtin* &Rest *Arglist*
Invokes the passed builtin (passed as a symbol) on the passed arglist, but adds an additional (first) argument: a *continuation point that will cause continuation at this level (invoked on failure).* Returns whatever the builtin does.

Invoke-Non-Deterministic-Builtin *(Builtin* &Rest *Arglist)* &Body *Success-Code*
Invokes the passed builtin (passed as a symbol) on the passed arglist, but adds an additional (first) argument: a continuation point that will cause continuation at this level (invoked on failure). Returns whatever the builtin does. This is similar to Invoke-Deterministic-Builtin, but on success, the Success-Code is run. Note that locals available inside of Invoke-BC-Protecting-Unbound-Globals are also available in the Success-Code.

[3]As with Invoke-Continuation, this is actually a function to aid in debugging, and should eventually be declared inline

Invoke-FC-Protecting-Unbound-Globals *((Function* &Rest *Arglist) Globals Backtrack Justification* &Key *(Generator '*Next-Value) (PGlobals '*Protected-Globals) (Result '*Current-Result) (Deterministic '*Deterministic) (Continuation '*This-Failure-Level) (Failure-Code NIL))* &Body *Success-Code*

This calls the function on the arglist passing two extra (first) arguments: an appropriate continuation point on failure, (that cannot be traced to a rvariable) and the continuation point set up for this level, used to identify which rvariables the function may have bound internally. The function will return a generator which can be used to get things appropriately bound in the globals-list for what constitutes a proof. Unbound globals are given a where-bound value of the continuation this macro generates. The idea is that if that rvariable later turns out to be a culprit, we can non-locally get back to the continuation generated by this macro. If the Func cannot generate *any* proofs, we invoke the passed Backtrack-Point. If we do generate some proof, control passes to the body of the macro. We bind local rvariables that are valid within the body: *THIS-FAILURE-LEVEL which is the continuation we generate, and *NEXT-VALUE which is the generator the Func returns. *Protected-Globals which is a list of all rvariables we protected, *Current-Result which is what the generated returned on invocation, and *Deterministic, which is non-nil if no further values are reasonable. Note that the caller can use his own rvariables for any of these should he prefer. After the Success code, we try the next value. Eventually we fail and invoke the Backtrack-Point (after invoking the Failure-Code if supplied). If the Backtrack-Point is NIL, we fall through.

Invoke-BC-Protecting-Unbound-Globals *((Function* &Rest *Arglist) Globals Backtrack Justification* &Key *(Generator '*Next-Value) (PGlobals '*Protected-Globals) (Result '*Current-Result) (Deterministic '*Deterministic) (Continuation '*This-Failure-Level) (Failure-Code NIL))* &Body *Success-Code*

Unsurprisingly similar to **Invoke-FC-Protecting-Unbound-Globals** in effect, though unfortunately dissimilar in execution. Mainly, this is due to FC and BC using the stack differently. BC returns a value, then must futz around reexpanding the stack to return another value, and FC doesn't need to return anything, so it is more efficient: when the stack is most expanded, it will add it's chained form.

Repeat-Invoke-Protecting-Unbound-Globals *(Generator Culprit Globals This-Failure-Level Protected-Globals Justifications* &Key *(Result '*Current-Result) (Failure-Code NIL))* &Body *Success-Code*

Very similar to **Invoke-BC-Protecting-Unbound-Globals**, except that macro is to initially do a proof, while this one is to get a subsequent value given an existing generator, and passing a Culprit. Virtually all of the arguments to this function are things that should have been saved from calling that one.

Barf-On-Culprits *Culprit*
Given a culprit, this function does the right thing to unwind the system to the state it was in at assignment to the culprit (thus it will potentially get another value).

Chapter 6

Dictionary of functions useful for Lispfns and Builtins

Note that while we are documenting many internal functions here which are useful or needed to write lispfns and builtins for Rhet, we make no claims to the stability of these functions between releases, or even patchs. That is, unlike the things that are documented in the User's Manual, these are really *internal* functions, and therefore subject to change with minimal notice.

These functions listed here are not meant to be exhaustive, but merely the set of funcitons from the reasoner and other packages that are most likely to be immediately useful when writing a builtin or lispfn. If, for example, your builtin needs to use the unifier directly, you should refer to the unifier chapter of this manual for more information about calling the unifier; it will not be listed here.

Axioms-B:Copy-Axiom *Axiom*
Since variables are destructively bound, this is the canonical way to assure a unique copy.

Axioms-B:Freeze-Axiom *Axiom*

Takes a (BC) axiom and 'freezes' it. This will turn rvariables, *etc.* into unique symbols, s.t. the resulting list can be operated on like a Lisp list. Thus `[[A ?y] < [B ?x ?y]]` which is an axiom structure will become ((A *VAR*-y-123) < (B *VAR*-x-243 *VAR*-y-123)) where each are atoms in a list rather than structures.

Axioms-B:Thaw-Axiom *Frozen-Axiom*

Takes a frozen axiom (as produced by Freeze-Axiom) and thaws it: *i.e.* generates a BC-Axiom structure from it. The rvariables are guaranteed to be unique, so (Thaw-Axiom (Freeze-Axiom Axiom-Foo)) can be used as a mechanism for structure copies.[1]

Heq:Convert-Form-to-Fact *Form* &Key *Context Truth-Value*

This takes a form without rvariables, and for any subforms assigns canonical names to them (so references will work) and then Gensyms a symbol for the new fact and returns this fact-accessor. Add-Fact is NOT called on this new fact-accessor. Normally that will be the first thing called after this call. The Truth-Value defaults to the truth-value of the form being converted, and can be overridden with the keyword option.

Heq:Convert-Form-to-FN-Term *Form*

This takes a form without rvariables, and for any subforms assigns canonical names to them (so references will work) and then Gensyms a symbol for the new fn term and returns this accessor. Add-Fn-Term is NOT called on this new fact-accessor. Normally that will be the first thing called after this call. The Truth-Value defaults to the truth-value of the form being converted, and can be overridden with the keyword option.

Heq:Simplify-Form *Form* &Key *Context As-Fact*

The most simplified form (*e.g.* with canonical names substituted as appropriate) is returned. If Form has any rvariables, an error is signaled. If the Form can be resolved to a canonical name or fact, the

[1] In fact, this is the idiomatic way to do so.

accessor for same is returned. Note that the fact returned is not necessarily appropriate for adding via add-fact, since canonical names may appear in it's arglist. Use **Convert-Form-To-Fact** instead. If As-Fact is specified and non-**Nil**, then Fact Accessors are prevented from being coerced into Canonical Name Accessors for the return value, since certain callers prefer the Canonical Name.

Heq:Type-Restrict-Can *Canonical-Accessor Type-Struct Context*
Updates the type of the passed Canonical to be type Type-Struct (an **ITYPE-STRUCT**) in context Context.

Heq:Type-Restrict-FN-Term *Function-Term-Accessor Type-Struct Context*
Updates the type of the passed Function Term to be type Type-Struct (an **ITYPE-STRUCT**) in context Context.

Hname:Accessible-HN *Itype-struct* &Key *Head Hash-Index Defaultp Index Context Recursive*
This is like **List-HN**, but will return all facts that are accessible to the current (or specified) Context. Thus, it's like a recursive **List-HN** for all parents of the Context.

Hname:Find-Fact *Head Arglist* &Key *Context*
This returns the accessor for the interned fact if it can be found. As a second value it returns the type (as an ITYPE-STRUCT structure), and as a third value, it returns the truth value (so it need not be looked up). Note that the arglist can only consist of fact-accessors or canonical-name accessors, function terms must be converted to canonical names. The accessor is what should be used to pass to **Remove-fact** as the Fact-Accessor, or to reassert a fact in a child context to shadow a parent fact. This function will look find a fact accessible to the passed context, rather than strictly in the passed context[2].

Hname:Find-FN-Term *Head Arglist* &Key *Context*
This returns the accessor for the interned function term if it can be found. As a second value it returns

[2]It is possible for Find-Fact to return a canonical name, for example, if the passed arglist consists only of canonical accessors, it is likely that a canonical accessor is interned in the hashtable.

the type (as an ITYPE-STRUCT structure). Note that the arglist can only consist of fact-accessors or canonical-name accessors, function terms must be converted to canonical names.

Hname:Get-Canonical-From-FN-Term *FN-Term-Accessor Context* &Key *Localp*
This function takes a Function Term Accessor and a Context and returns the canonical name for the fact in the context, or **Nil** if none exists. If the Local argument is non-nil, treat not having a canonical name directly in the Context the same as not having one at all.

Hname:Get-Predicate *Thing* &Optional (Context Reasoner:*Current-Context*)
Given a Fact accessor, or a Form, this function returns the purported predicate, *i.e.* the atom that is the head. Thus for Thing: fact [F A B] whose head is 'F and form [F A B] whose head is a **Fact-Accessor** for a Fact whose head is F and args are **Nil**; both would return the symbol F from this function. Canonical Name Accessors will use their primary. Any other argument generates an error. We take Canonical Names and even **Faccessors** with Canonical Names so we can be called inside of certain other functions, though they are not strictly predicates.

Hname:Get-Type-From-FN-Term *Fn-Term-Accessor Context*
Given a FN-Term-Accessor and a Context, return the term's type in this Context, which is stored in the **FN-Term** structure if it doesn't have a canonical name, and in the canonical name if it does.

Hname:List-HN *Itype-struct* &Key *Head Hash-Index Defaultp Index Context Keep-Unbound*
List-HN returns a list of all *facts interned in the current (or specified)* context of type Type, with head Head, and with index Index. A type of **Nil** is considered most general, (more so than ***T-U-Itype-Struct***, as it bypasses the typecheck) as is an index or head of **Nil** (the defaults). If the Defaultp flag is not set, default facts are not returned. The Keep-Unbound key is only used internally. If Hash-Index is supplied, it is used to speed the search.

RE-to-DFA:Compatiblep *String DFA*
Compatiblep Returns T if the passed DFA, from **Convert-RE-To-DFA**, could have generated the String.

RE-to-DFA:Convert-RE-To-DFA *String*
This function converts REs to compiled DFAs. The RE is passed as the the String parameter.

Reasoner:Abort-Rhet *N L*
Abort Backward Cha. ing. Possibly needs to clear up FC too...

Reasoner:Chain *Fact* &K y *Context*
Called by the RLLIB package when a fact has been added by the user. This invokes the forward-chaining mechanism as needed. It may call itself recursively, or as the result of an **Hassert-Axioms** form. Note that memory used by FC goes in it's own area.

Reasoner:Constraint-Satisfy-P *Constrained-Rvariable Form*
Returns T if setting the constrained-rvariable to the Form is not a violation of the rvariable's constraints.

Reasoner:Disprove-Goal *Legal-Goal* &Key (Proof-Type :Simple)
Returns a proof of [NOT GOAL]

Reasoner:Generate-BC-Proof *Failure-Continuation Reinvoke-Continuation Arbitrary-Form*
For practical purposes, a Builtin. Returns a generator to prove a goal that internally handles the generators of individual clauses defined that unify with the goal.In order for cuts and such like to work, this is the "builtin" that is the interface to the BC axioms. This is to assure that whatever set of causes indexing may proscribe as appropriate for the current proof, the CUT form can interact with this code (stackwise) and do the appropriate cutting action. Also, this makes prove-based-on-mode a *little* simpler...rather than having to do the indexing and get the generators for the axioms itself, this routine returns one generator (which internally will switch between the appropriate indexed generators). Note that in a proof tree, this routine represents an OR node: no state needs to be saved from prior things invoked; no caching is done at this level (only axioms, not builtins, can cache).

Reasoner:Interpret-BC-Axiom *Failure-Continuation Reinvoke-Continuation Axiom Goal*
Simulates the Axiom using the passed Rvariable bindings. This will recursively call the reasoner to

prove any subgoals of Axiom. If successful, it returns a first value of T and a second value of the new rvariable bindings. If unsuccessful, it's first value is **Nil**. This function is used by the reasoner to handle interpreted (vs. compiled) BC axioms. This is a prerequisite to using the stepper, and certain advanced trace facilities. All work is done in the ***Current-Context***.

Reasoner:Interpret-Builtin *Failure-Continuation Reinvoke-Continuation Form*
This function interprets a single clause in an axiom, which has already been determined to be a builtin. It returns whatever the builtin would return, normally 1. success or failure, and 2. the closure, if needed. All work is done in the ***Current-Context***.

Reasoner:Interpret-FC-Axiom *Continuation Axiom Trigger*
Simulates the Axiom using the passed Trigger. This will recursively call the reasoner only if a Prove form is encountered, to prove those subgoals of Axiom. If successful, it returns a first value of T and a second value of the rvariable bindings used. If unsuccessful, it's first value is **Nil**. This function is used by the reasoner to handle interpreted (vs. compiled) FC axioms. This is a prerequisite to using the stepper, and certain advanced trace facilities. All work is done in the ***Current-Context***.

Reasoner:Interpret-Lispfn *Failure-Continuation Reinvoke-Continuation Form*
This function interprets a single clause in an axiom, which has already been determined to be a user supplied Lisp function. It returns whatever the Lisp function returns. All work is done in the ***Current-Context***.

Reaonser:Lookup-Lispfn *Name*
Returns multiple values, the first of which is the argument symbol (to make lookup a predicate on a declaration for lisp-function-ness) the second is the Predicate-Function and the third of which is the Assert-Function as declared for function name by a **Declare-Lispfn** form. If no such form has been supplied for name, all values are **Nil**. Note that if the Assert-Function returned is **Nil** it means it was not supplied to the **Declare-Lispfn** form, and so the normal assertion mechanism should be used. The fourth value is a list of types that the arguments are expected to be subtypes of, for runtime typechecking purposes.

Reasoner:Prove-Based-on-Mode *Failure-Continuation Reinvoke-Continuation Goal* &Key *Context*
This does a prove simple or prove complex (*etc.*) depending on the call made originally by the user. It should be the common re-entry point to the reasoner for recursive proofs (*i.e.* subgoals). It returns a generator which is used to get the actual proofs. (Generator will return justifications for each proof). It uses ***FC-Active*** to tell if it is being invoked from FC, and if so, will not try to find BC axioms that prove the Goal.

Reasoner:RKB-Lookup *Continuation Arbitrary-Form* &Key *Context*
This function (generates a function that) unifies the arbitrary form against the KB, and returns the first fact-accessor that can match. As a second value, the closure to do this for the current arguments is returned. Backtracking is handled[3].

Rllib:Barf-On-Culprits *Culprit-List*
Given a culprit, this function does the right thing to unwind the system to the state it was in at assignment to the culprit (thus it will potentially get another value).

Rllib:Builtinp *Symbol*
Returns five values: The first is non-nil if the symbol is the name of a builtin predicate; it is in fact a string which is the name of the builtin as the user would call it from within Rhet. The second value is a list of the types of the arguments to this builtin (see **Define-Builtin**. The third is a keyword description of the type of the builtin, e.g. is it monotonic, foldable, *etc.*. The fourth is the symbol name of a function to be called if this builtin is asserted, as in the LHS of a FC axiom. The fifth is an UNDO function that is called to undo calls to a builtin with side effects.

Rllib:Complement-Truth-Value *Form-To-Complement*
This function takes a form and inverts the expected truth value. This is used by the prover to decide if it should look at facts like X or `[NOT X]` which is represented by the truth value on the fact. The form returned is EQ to the form passed, so the function is DESTRUCTIVE!

[3]or will be, anyway.

Rllib:Crunch-Vars-In-Arbform *Arbform-to-Crunch*

This function returns the arbform with any bound rvariables FULLY expanded. It takes pains to return a form that is EQ to the passed form, if nothing is changed, and a copy if something is changed so the passed form is not destroyed.

Rllib:Crunch-Vars-In-List *List-To-Crunch*

This function takes a list of arbforms and calls crunch-vars-in-arbform on each of them, and returns the new list.

Rllib:Generate-Bindings *Failure Variable Values-List*

Returns a function of one argument (the culprit) that will bind (via unification) the passed variable to each car of value-set. This function is intended to make writing builtins easier. See, for instance, the definition of **HMemberP**.

Type:Get-Frame *Type-Symbol*

Looks up the type's frame definition and returns the frame (an object of type **REP-Struct**).

Type:Get-Frame-from-Type-Hack *Type-Symbol*

Like **Get-Frame**, above, but since the user can give us a keyword or list for the type, and internally we want itypes, this returns the frame either way.

Type:Get-Result-Itype-struct *Function-Atom List-Of-Itype-structs*

The second argument should be a list of ITYPE-STRUCT structures representing the types of arguments to function-atom. The most specific type inferable of 'function-atom (**arg1 ...argn**)' where each arg has the itype-struct specified by argument-itype-struct-list will be returned. In particular if argument-itype-struct-list represents invalid types for function-atom, ***T-Nil-IType-Struct*** is returned.

The two primary ways Unifier can use 'get-result-itype-struct' are the following.

1. To see if `[any ?x*type1 [f ?x]*type2]` unifies with constant `[A]`, where **VAR** is a lisp variable whose value is the (constrained) rvariable structure, and **A** is the fact accessor for the constant...

After calling

(Typecheck (facc-type 'A) (Rhet-Term-Type VAR))

the Unifier should call

(Typecheck (get-result-itype-struct 'f (facc-type 'A)) (Rhet-Term-Type (car (var-constraints VAR))))

noting that **(car (var-constraints VAR))** is an oversimplified way of getting the **ITYPE-STRUCT** structure of the first constraint on **VAR**.

2. To see if ?x***type1** (call it **VAR**) and `[f ?y*type2]` (call it **FORM1**) can unify, the Unifier should call

   (Type-Exclusivep (Rhet-Term-Type VAR) (get-result-itype-struct 'f (Rhet-Term-Type (form-rvariables FORM1))))

   (which is again, an oversimplified example of how one would access the type of embedded rvariables in a form), just as the Unifier calls

   (Type-Exclusivep (Rhet-Term-Type VAR) (Rhet-Term-Type (form-rvariables FORM1)))

   to see if ?x***type1** and ?y***type2** can unify. If T is returned, the Unifier can detect failure. Otherwise, the Unifier can post constraints by producing `[any ?y*type2 [f ?y]*type1]` by calling (given **VAR2** is ?y***type2**):

   (Constrain-Var VAR2 (Create-Form 'RLLIB:HTYPE-QUERY (LIST FORM1 (Rhet-Term-Type VAR))))

   Note that in general, the resultant type may be a subtype of both input types — the Rhet unifier uses **TypeCompat** and sets the type of both objects appropriately.

Unify:Bound-Var-In-Goal-P *Form* &Key *Bound-In*
A predicate that returns non-nil if the goal (usually a form) has any rvariables bound. If Bound-In is supplied, only rvariables with that as a Where-Bound field are candidates.

Unify:Clear-Binding *Rvariable* &Optional *Constraints*
Resets the binding and optionally the constraints of the passed Rvariable to NIL.

Unify:Clear-Some-Bindings *Variable Binding-Location* &Key (Test #'Continuation->=)
Unbind the Variable if it is bound at the Binding-Location, as well as any constraints asserted at the Binding-Location. It returns the Variable again.

Unify:Clear-Some-Bindings-In-Form *Form Binding-Location*
Clear rvariables bound at Binding-Location that occur in the Form.

Unify:Constrain-Form *TermA Ground-TermB Function-Name*
This function knows that TermA contains vars, and TermB is ground. It constrains the vars in TermA such that Function-Name will be true between TermA and TermB

Unify:Constrain-Unbound-Vars *Form-to-Constrain Constraint-Form*
Puts a constraint on all unbound rvariables in the passed Form.

Unify:Constrain-Var *Var-To-Constrain Constraint-Form*
This function takes a rvariable that must be constrained such that Constraint-Form holds.

Unify:Continuation-> *Cont1 Cont2*
Returns non-nil if the first continuation is 'greater' (occurs later) than the second.

Unify:Continuation-= *Cont1 Cont2*
Returns non-nil if the continuations are equivalent.

Unify:Continuation->= *Cont1 Cont2*
Returns non-nil if the first continuation occurs later or at the same (stack) level as the second.

Unify:Crunch-Vars-Into-Form *Form*
Returns a newly Consed form with any bound rvariables replaced with their bindings.

Unify:Fact-Unifies-With-Form-P *Fact 1-Complex-Form Context*
Returns non-nil if the fact and form unify in the passed context.

Unify:FN-Term-Unifies-With-Form-P *Term 1-Complex-Form Context*
Returns non-nil if the term and form unify in the passed context.

Unify:Fully-Bound-Form-P *Form*
Returns non-nil if *all* the rvariables in the form are bound.

Unify:Get-Binding *Rvariable*
Returns the current binding of the passed Rvariable.

Unify:Last-Bound-Vars *VAR-LIST*
Given a list of rvariables, return the set of those rvariables most recently bound. Normally this will be used to determine a culprit if better info isn't available, the last bound var is the one to invoke.

Unify:Unbound-Vars-In-Form *Form*
Returns a list of all 'unbound' vars in the passed form. This includes vars that are bound at a level on the stack 'above' the current one.

UI:Cons-Rhet-Axiom *LHS* &Rest *RHS*
Return a standard Rhet bc-axiom (unasserted) given a list representation, *e.g.*

```
(L          et ((var-x (Create-Rvariable "?X")))          (Cons-Rhet-Axiom (Cons-Rhet-Form 'P var-x)
[[P ?x] < [Q ?x] [R ?x]]
```

UI:Cons-Rhet-Form *Head* &Rest *Arglist*
Return a standard Rhet form given a list representation, *e.g.*(Cons-Rhet-Form 'P 'A) returns [P A], (Cons-Rhet-Form 'P '(A B)) returns [P (A B)]. If the head looks like a builtin, it will be handled appropriately. Rhet variables should be created with Create-Rvariable and EQness will be preserved between calls to this function if the same Rvariable structure is passed.

UI:Real-Rhet-Object *Thing*
Returns the Rhet object associated with Thing, *e.g.* an accessor. Since accessors are the normal way

Rhet passes its structures around, this is a convenient function for the lispfn that wants to print its argument(s). **Rprint**, for example, uses this function.

UI:UI-Indexify *Index*
If the symbol is non-nil, return an index string built from that symbol, else return the default index string.

UIC:Archive-and-Return *Function Args Type Result*
Do the work of dibbling a call to Function on Args with Result. Type is either **:assert** or **:query**, and indicates the type of function being called (only one may be recorded, as per **Rhet-Dribble-Start**.

UIC:Copy-Goal *Goal*
Since variables are destructively bound, this is the canonical way to assure a unique copy.

UIC:Create-Form *Head Rest* &Optional (**Truth-Value :True**)
A standard fn for building forms, which will update the form-rvariables field. It is a useful interface to **Hname:Make-Form**. To convert lisp atoms directly to a form structure, use **UI:Cons-Rhet-Form** instead.

UIC:Create-Rvariable *Pretty-Name* &Optional (**Type *T-U-ITYPE-STRUCT***)
A function to construct variables. The Pretty-name is a string that will be the printed appearance of the variable. As such, it should begin with the character "?".

UIC:Find-Form-Rvariables *Form*
Calculates the **Form-Rvariables** field for a form by finding any rvariables and returning an list of the rvariables mentioned. Normally, one would use **Create-Form** which returns a form with this calculation made.

UIC:Find-or-Create-Term *Head* &Optional *Arglist* (**Type *T-U-Itype-Struct***)
If the appropriate function term is already known, it is returned. Otherwise, it is created and interned.

UIC:**Freeze-Goal** *Goal*
Takes a legal goal and freezes it. Interface to Freeze-LFP.

UIC:**Get-Type** *Arbform*
Gets the type of an arbitrary object and returns it.

UIC:**Make-I-Type** <*Type Symbol or List*> &Optional *Permissive*
Converts the passed lisp symbol into an Itype-Struct. If a list is passed, it is treated as a type specification as would appear after the astrick on a variable. This function is typically used, for example, by builtins that expect an argument to be a type specification. If Permissive is NIL, the default, return NIL if any component type is undeclared.

UIC:**Rhet-Equalp** *Term1 Term2*
Returns non-nil if the two things are CL:EQUALP or Rhet terms that are equivalent (not in the unification sense). If we (Copy-Goal [Foo ...]) the result should be Rhet-Equalp to the original.

UIC:**Set-Argument-Itype-Struct** *Form* &Optional (Result-Itype (Rhet-Term-Type Form))
Given a form and a desired resultant type, updates the types of arguments as necessary to guarentee a result that is that type (or a subtype of it). It returns the destrutively modified form if something was changed.

UIC:**Thaw-Goal** *Frozen-Goal*
Takes a frozen goal and thaws it. Essentially an interface to Thaw-LFP.

UIC:**Update-Form-Type** *Form* &Optional *Force-Type*
Destructively modify the passed form with a new type calculated from the Form's head and arguments, unless Force-Type is set, in which case use that and constrain the subforms appropriately. Be intelligent about forms that you can't calculate. Returns the Form.

UIC:**Warn-or-Error** *Item* *Checklist-symbol* *Continue-Control-String* *Proceed-Control-String* *Format-Control-String* &Rest *Format-Args*

Like Warn, returns non-nil if the user wishes to continue. A nil return implies failure; the calling builtin or lispfn would normally invoke a failure continuation. A bit snazzier than just WARN or CERROR, this function does a warning, and then asks if the user wants to go ahead (continue-control-string), error out, or go ahead and not ask again. Item and checklist is supplied by the caller for just this functionality: if item is found on checklist, Warn-or-Error will return non-nil; the user had previously indicated they wanted to Proceed on this sort of error. Checklist symbol is passed as a symbol; the standard place to put simple things (where the Item is a hardcoded keyword) would be on the **UIC:*General-Warning-List***, however the user is free to Defvar their own Checklist. They should then add an initialization to clear it to the **UIC:*Warn-or-Error-Cleanup-Initializations*** initialization list.

Chapter 7

High-Level User Interface

7.1 Overview

The Rhetorical system window interface is an interactive semi menu-driven program that allows the user to interactively use the Rhet System. It consists of a frame with several configurations (currently) and a process, associated with the frame. The frame is selectable via <select>-r.

For the most part, this code is so machine dependent it is not further documented here. You need to be a systems hack to make changes to this for a non lispmachine, since it is based on the presentation model of the machine. The symbolics version makes heavy usage of presentation types, and uses the **Define-Program-Framework** style of defining windows and commands. The explorer doesn't have a command processor, so instead it uses constraint frames and menus.

7.2 The Tracing Facility

Right now it doesn't exist, but eventually there will be hooks into the compiled and interpreted versions of axioms to allow simple tracing, along the lines of what a lispm does with lisp code.

Chapter 8

The Reasoner

This package determines the strategy for the proof of a particular goal or subgoal, picks axioms and facts as appropriate for the proof, and implements the horn clause semantics for many functions. (Some horn semantics are wired into the compiled axioms!). It uses the unification subsystem to bind arguments, *etc.* as necessary for interpreted proofs. Like most PROLOG compilers [Warren, 1977a] [Warren, 1977b] [Kahn and Carlsson, 1984] the Rhet compilers generate code with specialized unification code, rather than using the more generalized unifier provided by the Unification Subsystem[1].

The Reasoner will call Lisp functions as declared, rather than attempting a proof using the usual resolution mechanisms. Note that the ability to POSTpone evaluation of axioms (that is, to give a rvariable constraints) is implemented in this package.

Note that if default reasoning is enabled, the Reasoner will return proofs as before, but it may also have used some default rules in the proof - see ***Proof-Defaults-Used***. No consistency check is made on the defaults used in a particular proof.

[1]Which the interpreter, as well as builtins, in fact, use.

8.1 Its Flags and Functions

FC-ACTIVE Non-**Nil** When FC is active, so builtins can tell, basically.

FORWARD-TRACE This rvariable is set to the list of all assertions made via the **Chain** function. It must be reset by the user interface to clear it.

POSSIBLE-AXIOMS This is used as an interface between **Generate-BC-Proofs** and **Cut**. It is bound to the alternate axioms at the current level; **Cut** will clear out other possibilities.

PROOF-DEFAULTS-USED This is set to the list of default facts that were used or gotten via a default axiom in the current proof. It will be a list of lists if a **Prove-All** was used.

PROOF-TRACE This rvariable is set to the steps in the proof of the last full goal (in chart form).

REASONER-DISABLE-EQUALITY If non-nil, the Reasoner will refuse to use the equality subsystem in order to solve a proof. This gives the effect of a more pure PROLOG like language, with typed rvariables.

REASONER-DISABLE-SPECIAL-REASONERS If non-nil, the Reasoner will not take advantage of any declared specialized reasoners.

DISABLE-GOAL-CACHING If non-nil, RHET will not cache proved goals and subgoals. Normally this would only be used to debug the caching software.

REASONER-DISABLE-TYPECHECKING If this is non-nil, all type information is ignored during the proof process. Typed rvariables are sti l syntactically allowed, but treated as untyped.

REASONER-ENABLE-DEFAULT-REASONING If non-nil, the Reasoner will use default axioms or facts.

REASONER-PAUSE-FUNCTION If bound to some function, the function will be called between proofs during a **Prove-All**, **Prove-Complete**, *etc.* If it returns **Nil**, a Lisp abort is signalled. This is to allow the user interface to support doing only a limited number of proofs, or querying the user between proofs to see if he wants to continue. It is called with the forms already proven (with bindings).

REASONER-STEP-FUNCTION If bound to a function, it is called between proof macro steps. If it returns **Nil**, the proof is aborted. This is to allow the user interface to insert a break after so many steps of the Reasoner, as it did in HORNE [Allen and Miller, 1986]. It is called with the form to be proved (with current bindings) at the current state of the system as it's only argument.

TRACE-FORWARD-ASSERT If non-nil, each chained assertion is printed on the error output.

TRACE-REASONER If non-nil, a diagnostic for each proof decision is printed on the error output. If it's value is **:BUILTIN**, builtins are traced as well.

The exported functions are as follows:

Prove-Simple-B *Arbitrary-Form* &Key *Context*
Returns **Nil** if no proof is found via backward chaining from the goal arbitrary-form, using horn clause semantics. Otherwise, it returns the arbitrary-form (with rvariables bound, and with constraints, if any)

Prove-Simple-All-B *Arbitrary-Form* &Key *Context*
As **Prove-Simple-B**, but returns all distinct proofs that result in different rvariable bindings.

Prove-Default-B *Arbitrary-Form* &Key *Context*
Exactly like **Prove-Complete-B**, except that the proof/disproof only occurs at the topmost level, rather than in the entire proof tree.

Prove-Default-All-B *Arbitrary-Form* &Key *Context*
Exactly like **Prove-Complete-All-B**, except that the proof/disproof only occurs at the topmost level, rather than in the entire proof tree.

Prove-Complete-B *Arbitrary-Form* &Key *Context*
As **Prove-Simple-B**, but will also attempt a reverse proof using **Not** forms. Returns **Nil** if `[NOT arbitrary-form]` can be proven, **:Unknown** if no positive or negative proof succeeds, and **:Inconsistent** if form can be both proven and disproven. Note that while this may seem extremely inefficient, in fact goal caching is presumed to make this less of a problem.

Prove-Complete-All-B *Arbitrary-Form* &Key *Context*
As **Prove-Complete-B**, but returns all distinct proofs that result in different rvariable bindings, if the forward proof succeeds.

The important unexported (internal) functions are as follows:

Enqueue *Axiom-Access r Trigger Context Rank Queue*
Enqueues the Axiom-Accessor and Trigger for context Context on Queue with Rank.

FC-Process-Queues *NIL*
Processes the FC queues in priority order. That is, if there is an entry on the PURE queue, it is processed, otherwise the IMPURE queue, otherwise the BC queue. Note that successful axioms will CHAIN their result, so we must check all the queues each tir e. They may have changed! And, since we don't want the stack getting too deep, we return if we are not the top-level queue processor.

Global-Var-P *Var Unbound-Globals*
Returns non-nil if Var is a global, in the PROLOG sense.

Invoke-FC-Axiom *Axiom-Acc ssor Trigger Context*
Invokes the Axiom that was triggered via Trigger in context Context. The Axiom may or may not be compiled.

Invoke-BC-Axiom *Failure-Continuation Reinvoke-Continuation Axiom Goal Context*
Invokes Axiom using goal as the invoking goal in context Context. Axiom is expected to return a generator.

Pause-Check *Results*
Tests to see if *Reasoner-Pause-Function* exists, and if so, calls it.

PBM-FC-Link *Failure-Continuation Reinvoke-Continuation Legal-Goal*
This function is the builtin PROVE for FC. It maps into a BC proof, after setting up things right.

Recursive-Interpre.-FC-Clauses *Axiom RHS-Clauses-Left Justifications Failure-Continuation*
This is the guts of interpreting clauses. We have to do it this way to get continuations to work right, since we can only continue to things that are active (a pity, too). RHS-Clauses-Left are the uninterpreted clauses of Axiom. We collect proved forms on the Justifications list. If we can't prove the Car of RHS-Clauses-Left, we invoke the Failure-Continuation.

Recursive-Interpret-Bc-Clauses *Failure-Continuation Unbound-Globals-Continuation Left-To-Prove*
Does the gruntwork of BC proofs: proves the next clause on the RHS of the Axiom we are proving, saving continuations, *etc.*, for restarting on internal closure rvariables. Note that we don't have to return anything, the destructive bindings of axioms will do it all. In fact we return the justification for whatever was proved. This function in reality returns a generator.

Uncrunch *LIST*
Takes a justification list and strips out anything that isn't a fact accessor.

8.2 Its Design

The Reasoner implements a somewhat enhanced proof procedure compared to the old HORNE system. That is, given some subgoal, it will first see if it, or something equivalent to it is asserted (unless it is a Lisp function), and if so it provisionally succeeds. If not, it checks to see if the inverse of the goal is asserted (or something equivalent to it), and if so fails. If not, it will attempt to prove the axiom. If it cannot prove it, it will attempt to prove the inverse (if the appropriate function was called from the QUERY interface), and if it still can prove nothing it will return the fact that it can neither prove nor disprove the goal, rather than simply fail.

Note also that the Reasoner may invoke the equality subsystem itself rather than relying on the Unifier to do it, should rvariables not appear in either expression, or for other reasons depending on the wired in heuristics.

The heuristics the Reasoner uses to determine how it goes about attempting to prove or disprove some subgoal *should be made easily modifiable*, so experience with the Reasoner will allow improvement in these heuristics without a major rewrite. In fact, right now it is concentrated in **Generate-BC-Proof**, though **Prove-Based-on-Mode** and the interpreter have something to say about it. (the compiler will probably have more).

FC axioms that use the PROVE form get the first proof via BC. Backtracking of the high-level PROVE form is NOT supported. Further, this form is assumed not to have side effects. Similarly for builtins, *etc.* in the RHS of a FC axiom.

Rvariables are bound directly, and destructively, in the rvariable structure [Bruynooghe, 1982] [Mellish, 1982] [Kahn and Carlsson, 1984] [Sterling and Shapiro, 1986]. All internal functions and builtins handle arguments that have this structure, such that intelligent backtracking [Warren, 1986] [Bruynooghe and Pereira, 1984] [Cox, 1984] can be implemented. The idea is that all functions that can backtrack take a failure continuation as an argument. If they cannot find any proof for the form handed them, they invoke the continuation. If they can identify a particular *culprit*, however, a slot on the rvariable contains the continuation that should be invoked. For example, given the following: `[[P ?x] < [Q ?y] [R ?x ?y]]` if we use naive backtracking, given the goal `[P A]` if we can prove `[Q B]` we will try to prove `[R A B]`. *If that cannot be* proven (possibly after much work), we will backtrack and reprove `[Q ?y]` for some other value, possibly `[Q C]`. Intelligent backtracking allows us more latitude: if during the course of proving `[R A B]` we determine that `[R A ?z]` would have failed (*that is there is no proof of predicate* R with it's first argument bound to A) we call ?x the *culprit*. Since in our rule ?x is a local, it does no good to reprove anything else in the rule, the rule can immediately fail. Should the rule have been *deterministic*, that is, the "last" possible way to prove `[P A]`[2] then we can in fact invoke the last backtrack point... possibly all the way up the stack to the point were A was bound. A more complete example: Let's say that this was our only rule to prove P, and it was

[2]either it is the only rule for proving `[P A]` or all other rules to prove it have failed

invoked as a subgoal of the rule: `[[Z ?x] < [P ?x] [T ?x]]`; further that is the only way to prove Z; and it was invoked by the rule `[[GOAL ?x] < [R ?x ?y] [Z ?y]]`. `[GOAL T]` was the thing the user asked the system to prove, and the KB has `[R T A]`, `[R T C]`, `[R C F]` and `[Q B]` `[Q F]` in it. This binding of ?y to A is the culprit by our logic above, so once we determine that we will immediately retry that subgoal, getting the new binding of C.

In order to make all this work, we internally do two things: proving something via FC or BC is normally finding some goal, and getting a closure which is used as a generator to give us successive proofs of the goal (possibly with different global rvariable bindings). For example, when we call RKB-LOOKUP on `[R ?x ?y]` we might get ?x/T ?y/A first, and on reinvokation ?x/T ?y/C, *etc.*. Similarly, if there were an actual RULE that did this proof, we would find it by the usual mechanisms and get back a closure: the interface to an axiom (compiled) that does BC proofs of some goal is really identical to the interface to RKB-LOOKUP, the only difference is that the latter is more efficient, it just checks the KB. Once a generator has proven (or failed to prove) some goal, we cache it [Fagin, 1984] so we can reuse the result elsewhere in the proof tree as needed[3].

See the library section 10 for a description of the various macros that are defined to manipulate these closures such that the failure continuations work correctly.

8.3 Unimplemented

Indexing is currently much too trivial (just on head), but that's all you get until I have time to work on it. What's there should be good enough for testing and getting things working.

Compiler just compiles a call to the interpreter in.

Justifications kept are currently a hack: either the fact accessor or a cons of the *builtin/Lispfn and rvariable bindings*. Jun actually just wants the fact accessors, but right now I'm using it for debug. Trivial enough to pass less.

[3]Well, we will, it isn't implemented yet.

Chapter 9

The Axiom Database Subsystems

The axiom database keeps the forward chaining and backward chaining axioms segregated, although they are stored similarly[1]. Note that where triggers, LHSs *etc.* are supplied, these are to be **Forms**. These are somewhat restricted: typically triggers, and LHSs that are supplied must have their **Form-Head** bound to a fact-accessor, rather than an arbitrary term (*e.g.* a **Rvariable**, another **Form**, *etc.*).

Contexts are more restrictive in axioms than they are in facts[2]. The main restriction is that, unlike facts, axioms cannot be deleted in a specific context, nor can they be shadowed with an alternate form[3]. Axioms

[1]see section 9.2.

[2]This may be something worth discussing - we didn't see a particular need to support something more, which could, of course, be done, but would be harder. Just to be able to make a particular axiom invisible from some child on down would not be too hard - basically involve adding a 'lower' bound context list to the axiom structure, though it would make lookup more cumbersome. We would not need the complex approach that was taken with facts, since axiom lookup is much less frequent; we can afford to be a little inefficient for the sake of space and code complexity. Second, if needed, the prover is the real mechanism that would need to be fast here, and the hashtables it uses COULD be copied into and updated for specific contexts if necessary.

[3]Again, we thought such things unnecessary. It is even unclear how they will be used for facts: they exist as a side effect of

that are asserted to a context are accessible to all child contexts. To prevent an axiom from being accessible in a particular child, it would have to be removed from the parent.

9.1 Using Them

Here are the flags and functions that are exported as they are currently defined, subject to change, *etc.*

Freeze-Package When freezing an axiom, what package the frozen symbols are interned in.

Frozen-Var-Htable A hashtable of the rvariables that are frozen for the current form.

Note: All functions take an optional keyword *argument* **:Context**, *to specify the context to be used for the* command. The default used is the value of ***Default-Context***, since these functions may be called from the *user interface.*

All of the following functions *return a list of the axiom accessors* they operated on[4]. Those that take a defaultp flag will only operate on axioms marked default if it is set. The removal functions ignore the default flag on the axiom. For example, an axiom added via **Add-Axiom-B** with Defaultp set to non-**Nil**, will not be returned by an appropriate **List-All-Axioms-B** even if it matchs, if the List function is called with Defaultp Nil. But a remove function that matches will remove this axiom, since remove functions ignore the Default flag.

Add-Axiom-F *Axiom* &Key *Context Defaultp Compile*

Adds axiom to the forward chaining database, after doing consistency checking on it. This may involve compiling the axiom[5] Defaultp is by default NIL (non-default chaining rule). A non-nil value will make

other decisions.

[4]I hope.

[5]The Compile key is by default, T. Specifying it to be nil will prevent compilation, a necessary precursor to using the stepper

the axiom marked as DEFAULT, and only used when default reasoning is enabled. Implementation note: Note that this function need not be called to add an already compiled axiom, such could be "dropped" directly into the KB when the compiled file was loaded. Thus, care must be taken in the axiom compiler to appropriately wire in the compiled form into any structures. Returns the new axiom accessor.

Add-Axiom-B *Axiom* &Key *Context Defaultp Compile*[6]
Adds axiom to the Backward chaining database, after doing consistency checking on it. This may involve compiling the axiom. Defaultp is by default **Nil** (non-default chaining rule). A non-nil value will make the axiom marked as DEFAULT, and only used when default reasoning is enabled. Implementation note: Note that this function need not be called to add an already compiled axiom, such could be "dropped" directly into the KB when the compiled file was loaded. Thus, care must be taken in the axiom compiler to appropriately wire in the compiled form into any structures. This function always returns T, as any error is signaled. Returns the new axiom accessor.

Remove-Axiom-F *Trigger* &Key *Context*
All axioms that match the literal trigger and are in the passed/default context are removed from the database. Returns a list of the axiom accessors.

Remove-Axiom-B *Axiom* &Key *Context*
All axioms that match the literal form (on the left hand side of a horn clause) and are in the passed/default context are removed from the database. Returns the list of the axiom accessors so removed.

List-Axioms-F *Trigger* &Key *Context Defaultp*
The axiom accessors for all axioms that match the literal trigger are returned. Unless Defaultp is specified to be T, axioms added with Defaultp NIL are NOT returned.

List-Axioms-B *Form* &Key *Context Defaultp*
The axiom accessors for all axioms that match the literal form are returned. Unless Defaultp is specified to be T, axioms added with Defaultp NIL are NOT returned.

[6] as above

ɪ st-All-Axioms-F *Atom* &Key *Context Defaultp*
All axioms with atom as the head of a trigger have their accessors returned. Defaultp works as for List-Axioms-F.

List-All-Axioms-B *Atom* &Key *Context Defaultp*
All axioms with atom as the head of their left hand side have their accessors returned. Defaultp works as for List-Axioms-B.

Remove-All-Axioms-F &Key *Context*
All axioms directly in the passed/default context are removed from the forward chaining KB, whether added via Add-Axiom or directly by loading a compiled file. Returns no value.

Remove-All-Axioms-B &Key *Context*
All axioms directly present in the passed/default context are removed from the Backward chaining KB, whether added via Add-Axiom or directly by loading a compiled file. Returns no value.

List-Axioms-By-Index-F *Index* &Key *Context*
All axioms with matching index have their accessors returned.

List-Axioms-By-Index-B *Index* &Key *Context*
All axioms with matching index have their accessors returned.

Remove-Axioms-By-Index-F *Index* &Key *Context*
All axioms with matching index and in the passed/default context are removed from the forward chaining KB. A list of the axiom accessors is returned.

Remove-Axioms-By-Index-B *Index* &Key *Context*
All axioms with matching index and that are in the passed/default context are removed from the backward chaining KB. A list of the axiom accessors is returned.

Compile-Axiom-F *Axiom-Accessor* &Key *Context Defaultp*
Called internally by Add-Axiom-F, it is provided so the user interface can provide a function compiler and dump forms as necessary to a BIN file. Implementation note: part of compiled form may be an **(eval-when :load)** so when the form is loaded it will wire itself appropriately into the KB. The user interface is responsible for Remove-Axiom-Fing the old definition of a function that is changed by the user. See Add-Axiom-F for a description of the Defaultp flag.

Compile-Axiom-B *Axiom-Accessor* &Key *Context Defaultp*
Called internally by Add-Axiom-B, it is provided so the user interface can provide a function compiler and dump forms as necessary to a BIN file. Implementation note: part of compiled form may be an **(eval-when :load)** so when the form is loaded it will wire itself appropriately into the KB. The user interface is responsible for Remove-Axiom-Bing the old definition of a function that is changed by the user. See Add-Axiom-F for a description of the Defaultp flag.

Generate-Key-From-Fact *Fact*
This function takes a fact and generates the appropriate key to use in finding appropriate axioms to use in it's proof or chaining. The head of the fact is not used in creating the key except indirectly, as this is normally passed as a separate parameter to Get-FC-Axioms-By-Index anyway.

Generate-Key-From-Fn-Term *FN-Term*
This function takes a function term and generates the appropriate key to use in finding appropriate axioms to use in it's proof or chaining. It's just like Generate-Key-From-Fact.

Generate-Key-From-Form *Form*
Given a ArbForm, return what we will use to match the index, and get a 'hitlist' of possible axioms to trigger. Very similar to Generate-Key-From-Fact.

Get-FC-Axioms-By-Index *Head Key Context*
The Index referred to by the name of this function is the hash index, not the axiom index. This is the Key parameter, and is calculated based on the trigger we are attempting to find an axiom to chain on.

Given the head and a good key, we try to pick only those FC axioms that are likely to fire on the trigger. We do this by doing stuff at compile-time, *e.g.* if the trigger is `[F [S ?x]]` we know that the argument to the head F must be a structure and we index our axiom in the axiom KB based on that. During a proof, if we are given an actual trigger of `[F :A]` we will calculate the head to be `[F]` and the key is an atom, which is not a structure, and thus we will not pick this clause as an appropriate one to fire.

Get-BC-Axioms-By-Index *Head Key Truth-Value Context*
The Index referred to by the name of this function is the hash index, not the axiom index. This is the Key parameter, and is calculated based on the goal we are attempting to find an axiom to prove. Given the head and a good key, we try to pick only those BC axioms that are likely to prove the goal. We do this by doing stuff at compile-time, *e.g.* if the goal is `[F [S ?x]]` we know that the argument to the head F must be a structure and we index our axiom in the axiom KB based on that. During a proof, if we are given an actual goal of `[F :A]` we will calculate the head to be `[F]` and the key is an atom, which is not a structure, and thus we will not pick this clause as an appropriate one to use.

The following functions are not exported. They should not be used, and are documented here for completeness and allow possible code-sharing in the future (if you know what is already written, you won't need to rewrite it!).

Get-Axiom *Axiom-Accessor*
Given the axiom accessor, return the axiom structure.

Print-FC-Axiom *FC-Axiom-Structure Stream Depth*
Supplied as the print function for FC axiom structures, it pretty-prints them.

Print-BC-Axiom *BC-Axiom-Structure Stream Depth*
Supplied as the print function for BC axiom structures, it pretty-prints them.

Ill-Formed-FC-Axiom-P *FC-Axiom*
This does some rationality checks on the FC-Axiom structure. Returns T if there is a problem.

Ill-Formed-BC-Axiom-P *BC-Axiom*
This does some rationality checks on the BC-Axiom structure. Returns T if there is a problem.

Index-Axiom *Form Axiom-Accessor KB-Name Context Defaultp*
This function indexes the axiom accessor into the KB-Name using the passed Form as the Index. It is used by both the FC and BC front ends, since they are indexed identically.

Trim-Unaccessible-Axioms *AA-List Context Defaultp*
Given a list of axiom accessors, this returns the list with unaccessible axioms deleted; that is, axioms that would not be accessible to the passed Context, or are marked as default when Defaultp is **Nil**.

Reorder-Axioms *Axiom-Accessor-Moved* &optional *Axiom-Accessor-Indexing*
This takes the axiom associated with the Axiom Accessor Moved, and moves it to before the Axiom-Accessor-Indexing for a particular predicate's proof order. To move an axiom to the end of the order, do not specify an indexing axiom accessor.

Find-BC-Axiom-Locals *BC-Axiom*
Given a BC axiom, figure out what the local rvariables are, and return them. Note that all we might think we have to do is return the rvariable list for the LHS, but that isn't sufficient: a rvariable could be for passing info upward on the LHS, which is effectively a global. That is, it will only be bound to a rvariable on invocation. Assume the compiler does this; worst case individual axioms have to do their best.

Find-BC-Axiom-Globals *BC-Axiom*
Given a BC axiom and the local rvariables therein, compute the globals used.

9.2 Figuring Out How They Do It

Implementation note: Contexts are not handled as nicely as with facts, because of the potential problems with ordering axioms in multiple namespaces. Relevant context is kept in the **Basic-Axiom** structure, and part of the axiom's program checks[7] it's own accessibility in the ***Current-Context***. If an axiom is NOT accessible, it fails quietly. (That is, it does not signal an error). Also, the Reasoner will not in general attempt to invoke an axiom that is not accessible[8], so this may not be strictly necessary.

Another implication of this handling of contexts is that, as described above, we only handle the case of an axiom being interned in some context and accessible to all children of that context (checked via ***Accessible-Context-P***). *This kept the code very simple, and lookup reasonable.* Something more complex is possible, at a concomitant cost in code complexity and time for usage.

All of the functions in **Axioms-F** and **Axioms-B** are quite straightforward. The basic plan is to store the axioms, compiled, in a package (either **FC-Axiom-KB** or **BC-Axiom-KB**) and invert them based on their index and either their LHS (for a BC axiom) or trigger (for a FC axiom) which is how the reasoner will typically want to look them up. Many functions operate on all of the symbols that have been interned in these packages (such as delete-all). They access the symbols one at a time and see if they meet the proper criteria. As facts, axioms are stored as an accessor (an atom) that is interned in the package. The value of this accessor is the axiom structure, the function-binding is the compiled form, or a (compiled) call to the interpreter, as needed. The heads and indexes are interned directly; their value is a list of the accessors they match.

[7]Yet to be implemented.

[8]although this requires extra work for the selection criteria mechanism, and would be best to be on the hash table in that case.

9.3 Future Work

Right now no mechanism is provided for changing the axiom order[9]. We plan on leaving this undefined to allow a future parallel implementation. The fact that axiom accessors were used (unlike fact accessors, axiom accessors are not intrinsically necessary) are a concession to the fact that we need some sort of tag or "proper name" for individual axioms, so such a reordering routine can be built. Since its complexity will be directly related to how we will in fact build up axiom structures into hashtables or lists for the reasoner (we will probably want to do things to make finding the applicable axioms and facts as fast as possible. Warren [Warren, 1977a] indexed not only the head of the LHS but also on the type of the next symbol: we may want to use the types of the entire LHS, even for facts.) Since this is still unclear, I have left the issue temporarily open, pending completion of the Reasoner and axiom compiler designs. We currently do not allow axiom reordering since we want to keep our options open for future parallel implementation on the BBNButterfly^{TM}, and then a defined axiom order would be a hindrance (similarly for RHS order!)

The Ill-formed axiom functions currently do nothing (always fail) pending future need, when the user interface will call some functions directly, or need something to check the user's horn clause for legality.

The axiom compiler only compiles a call to the interpreter on the passed axiom. This, until more can be done in the upper packages so support and format can be settled on.

Some hashing speeding access to the axioms depending on just what is to be unified with should be possible (*e.g.* if we will unify with `[P A ?x]` we should be able to hash to all axioms of the form `[P A ...]` and `[P var ...]`). Right now we only cut our search depending on the head, which we require to be a fact-accessor.

[9] Clearly this is only an issue for BC axioms, as all FC axioms whose triggers are matched will be fired, although the order of firing might be an issue to TMS.

Chapter 10

The Language Definition Library

10.1 Using the Library

Section 5.4 describes the macros and functions provided to support *builtins*.

The above, as well as other code, also use the following special vars:

Debug-Continuation-Establishment if non-**Nil**, it is expected to be a function, that is called with a format string and arguments each time a continuation is established.

Debug-Continuation-Compile-Flag Separates whether or not we compile in the debug code, and if it's active.

There are also functions associated with each of the builtins described in the User's manual. Since these simply do what was indicated there, documentation is not repeated for them here. Note that these lispfunctions that are defined as builtins need not have the same name as appears to the user. For example, **Hequalp** is the lisp function that handles the builtin EQ?, while the lisp function name and the Rhet builtin name for **Bagof** are identical. Each take two additional first arguments before their documented arguments in [Allen and Miller, 1989]: their first argument is a failure continuation, which is invoked when the builtin cannot successfully prove something, and a reinvokation continuation, which is passed as information to the builtin: it is the continuation that will be invoked to request an additional proof *from* the builtin. All of the builtins return a generator that supply successive "proofs" when invoked. Note that **Define-Builtin** declares if a builtin has side effects (and if so, what it's undo function is). All are expected to handle backtracking. (If it does not make sense for a particular builtin to backtrack, it still returns a generator, that will only generate one value then invoke the failure continuation if reinvoked).

10.2 Design Details

Right now, the MAC forms of the above, purely expand into the normal calls, with deferred calls to bind any rvariables "discovered" to be bound at actual execution time. This is unlikely to actually work, but the compiler won't actually be done for a while.

For the most part, this code is reasonably straightforward. All functions adhere to the interface of taking a continuations parameter, and returning a success/failure indication, and the closure if needed. A good proportion of the code assumes that by the time the basic functions are called, any rvariables have been expanded (replaced by their bindings), but this is changing to take advantage of intelligent backtracking. Functions which do not bind rvariables, will often not bother with a closure (since there is no backtracking to be done anyway).

Recursive proofs are typically done using the Reasoner:Prove-Based-On-Mode call, though FORALL which does complete BC proofs, uses the higher level functions.

Certain functions expect fact accessors, even if a canonical name might have been used. For example, given that `[A B]` and `[C]` have been asserted to be EQ, which is used for the **Hatomp** test to determine atomicity?

10.3 Left To Do

- It is unclear that the macros to provide compiled / compilable code will really work. Specifically, some things (like context) will come out with the current context's package, but to store this in a file wouldn't work.
- Several builtins don't work or aren't defined yet.
- It is incredibly hard to wr'te a working builtin, unless you copy some other canonical one. The macros need to be rewritten to ei er provide a general Schemeish continuation into inactive closures capability, or at least, some general "e vironment" macro that handles some of the macro-to-macro communication and unburdens the user needs done.
- The macros need enhanced to do the right thing if there are side effects.
- Caching needs to be handled right, particularly if there are side effects; want to know if caches must be flushed (and if so, which ones). This may involve enhancing define-builtin to advise how much of a flush is needed.
- Right now, builtins are not unified against, they are just called with the entire arglist. May want to:
  - match the arglist against the types declared to define-builtin, and complain if there is a problem.
  - break builtins that handle multiple argument types into separate builtins; the compiler/interpreter calls the appropriate version depending on arg types[1].

[1]Now, the builtins handle this problem internally. This may be only slightly less efficient than having the compiler precompute (if it can), and thus not worth the effort. Certainly maintenance of a particular builtin can be more of a chore so that implies some macro needs written to automate it.

Chapter 11

The Unification Subsystem

The job of the Unifier is to take two forms and calculate bindings for each rvariable needed to unify them. Equal rvariables in the two forms are assumed to be the same. Type restrictions of the rvariables are taken into account. With the structure copying approach, the unifier no longer returns the bindings, but rather the rvariables are destructively modified to contain their bindings as part of their structure.

11.1 The Usage

Note: All functions that take an optional argument for context, will default to the value of *Current-Context*[1].

The following globals are defined:

[1] *Not to be confused with* *Default-Context*.

CURRENT-CONTEXT This is initially set to the root context. It is the context the Unifier is currently working in, and may be reset by the Reasoner whenever a proof demands it. It is distinct from *Default-Context* since the user may be called to manipulate the KB during a proof.

Here are some of the functions the Unifier package provides (that aren't documented elsewhere):

Simple-Unify *Type Arbitrary-Form* &Key *Context*
Used when unifying with a simple typed rvariable. The function checks that the Arbitrary-Form is compatible with the Type (an **Itype-Struct**) and returns success indication, then Arbitrary-Form unsimplified as the second value, unless some simplification is necessary to check the type (as in the case of a function defined to return different types depending on the types of it's arguments). The function returns three values, the first is **Nil** if the type of the Arbitrary-Form is not compatible with Type. Note that the Arbitrary-Form can be a **Form**, a canonical name accessor, or a fact accessor. The *third value returned* is the most constrained type of the object and the type passed. Unless the object is a rvariable, intersection at type **T-Nil** is considered unification failure.

Complex-Unify *Arbitrary-Form1 Arbitrary-Form2* &Key *Context*
This function unifies two arbitrary expressions, using equality to simplify or support as necessary. The function may call itself recursively. The function returns **Nil** if no unification is possible between the two forms specified as its first value.

Unify-Without-Equality *Arbitrary-Form-1 Arbitrary-Form-2* &Key *Context*
Like **Complex-Unify**, but refuses to use the equality subsystem. Thus, if the two forms do not directly unify, this function returns **Nil**[2].

Generate-Bindings-Alist Rest *List-of-Forms*
Given one or more forms, the function returns an Alist of the rvariables mentioned and their bindings.

[2]This function is provided so the Reasoner package may support alternative proof strategies, and so the user can disable the equality system temporarily if desired.

Clear-Bindings-in-Form Rest *List-of-Forms*
For all forms passed, the bindings for all vars are cleared.

Unify-arbform *Arg1 Arg2 Context*
Attempts to unify two arglists of two different forms. Return value is a boolean indicating success. Arg1 and Arg2 are lists, Forms Rvariables, fact accessors, or canonical name accessors.

11.2 The Description

The fundamental initial algorithm, was from Wilensky's treatment in [Wilensky, 1986]. Unify-Form assumes it's arguments are Forms, while Unify-Arbform does not make this assumption. Unify-Form first attempts to unify the heads of the two forms (via Unify-Arbform) It then does a Unify-Arbform on the Car of the current arglists, and again on the Cdr.

Unify-Arbform is somewhat more complex. If one of its arguments is a Rvariable, it calls Rvariable-Match on the rvariable and it's other argument. If both arguments are Forms, it calls Unify-Form on them, and otherwise attempts to simplify any argument that happens to be a Form (via Simplify-Form). If it ends up with two fact or canonical name accessors, it calls Test-Equality on them. Otherwise if we are still dealing with lists of objects, we call ourselves recursively on our Car and Cdr.

Fact-Unifies-With-Form-P basically converts the call into a call on Unify-Arbform.

Rvariable-Match checks to see if the rvariable is already bound, and if it is, calls Unify-Arbform on the bound value and the item passed. (If it succeeds, we know it was an equivalent object so we are OK). Otherwise we bind the rvariable to the form, providing the types are consistent. To prevent problems we don't bind a rvariable to itself, we just succeed.

Contained-In makes sure we aren't, for example, attempting to unify ?x with `[A ?x]` which would otherwise recurse forever in some sense. It checks that the rvariable is nowhere present in the form, by recursion on the form if it extends over multiple levels. If it finds a rvariable in the form bound to the rvariable we are checking for, this also counts. This occurrence check is omitted, for efficiency, if *Omit-Occurrence-Check* is non-nil.

11.3 Left to Do

- The *Unifier* should handle sets, *currently it doesn't*. That is, a set should only unify with other identical sets.
- Since the type facilities aren't fully installed, there is likely to be further work to do in interfacing it. Specifically, we should be able to look up the type of forms we don't yet know the type of, if the user has declared such function terms (of arguments) to be of a type.
- Need to handle constructor functions as a special case.

Chapter 12

The Hierarchical Equality Subsystem

Interfacing to the Hierarchical Name Subsystem, functions in this package will add new equality assertions, or simplify fully grounded expressions for the Reasoner and the Unifier. It also provides equality-based retrieval to the Reasoner and Unifier, that is, sets of facts based on a given canonical name. This may include, for example, possible parameters to a function such that the function will then be equivalent to a given generic name. Implementation note: Note that there are no compiled forms specified for defining equalities. Because most of the work would have to be done on the load of the so-called compiled form, no advantage could be seen in providing it. Instead, the world save[1] is considered an adequate and much faster way to store equalities between sessions.

[1]A provision of the lisp machines, and certain other lisps, is the ability to snapshot the entire environment into a file, which can be restored at a later time.

12.1 Overview

This subsystem provides E-Unification to the Rhet system. E-Unification is further described by Kornfeld [Kornfeld, 1983], and is also taken up, for example, in [Haridi and Sahlin, 1984]. The algorithm Rhet uses to maintain equalities is fairly simple in concept, though more difficult in practice. The basic idea is that two terms that have been asserted to be equal, have the same canonical name. Rhet's algorithm, unlike, say, Union-Find (see [Hopcroft and D., 1979]) does all of it's work on assertion, in order to achieve O(1) lookup time for equalities. Thus, to check if two terms are equivalent, we merely have to get their canonical names, and see if they are the same. This is made somewhat more complex since the canonical name may depend on context, however, for some fixed number of contexts, lookup is still O(1)[2].

These canonical names that are blithely compared by unification, are actually accessors to canonical name *structures that are somewhat more complex*. The canonical name structure contains a list of all the equality terms that hold the canonical name, a primary element of that set (the one the user would be most likely to want to have presented for the set, *e.g.*, given that `[ADD-EQ [Father-of John] Fred]` we'd want the system to present `[Fred]` rather than `[Father-of John]` on output, so it would be made the primary. The canonical name structure also contains a list of equality terms that are referenced by this one. In our `[Father-of John]` example, `[John]` would be one of the references. Rhet has to track references since if `[John]` is asserted to be equal to something else, say `[Sam]`, then `[Father-of Sam]` would also be in `[Fred]`'s equality class. This is particularly important with contexts, since we may have, for instance, already have inconsistently declared in some child context that `[Fred]` and `[Father-of Sam]` are *not* equal! Or, further, `[Father-of Sam]` may already have some independent equality class, that must now be unioned in with `[Fred]`'s canonical class. Naturally, all of this must be tracked by context, so typically, if a union is to be done in a particular context, then all of the child contexts to it are processed first, if any of the canonical names are also aliased in some sense in one of the children. This is complicated by the fact that a canonical set in a particular context may

[2]The time to find the canonical name of a term is proportional to the time to run down an assoc list of contexts and canonical names that is associated with the term. Since a particular term typically has few canonical names, this is a reasonable solution. Should our typical case exceed about 25 or 30 names on the lisp machine, a hash table may become beneficial.

only be a small part of the set in a child context, since further unions may be effective there.

Last, rather than being careful to reuse the canonical classes as the standard Union-Find algorithm is, we always generate new classes, because this makes it easier to undo the union request, if during processing we detect that a recursive union would fail. This could happen either because of a direct inequality assertion, or because of an inconsistency with a predicate. An example of the former was presented above, and an example of the latter might be attempting to assert that `[Sam]` and `[John]` are equal, when `[Owns-Helicopter-P Sam]` has been asserted, as has `[Not [Owns-Helicopter-P John]]`.

12.2 Functions and Flags

ENABLE-HEQ-WARNINGS If non-nil, warnings will signal an error. Otherwise they are ignored.

Note: All functions take an optional keyword argument :Context, to specify the context to be used for the command. The default used is the value of *Default-Context* for the equality definition functions, and *Current-Context* for the equality test and lookup functions (normally called from the reasoner).

Define-Equality *Function-Term1 Function-Term2* &Key *Context*
Uses Def-EQ to assert in the current (or specified) Context, that two Function-Terms are equivalent, and of the appropriate implicit type. Note that if the type of the two simple Function-Terms are not compatible, an error is signaled. The function returns the canonical name of the the new class. Note: should one of the arguments already be in a class, the other is added to it. Should both be in different classes, they are collapsed (via HN-Union), and the surviving class name is returned.

Def-Eq *Canonical-Name Function-Term* &Key *Context*
This function will call HN-Union on the Canonical-Name and Function-Term after verifying that the type (implicit) of the Function-Term is consistent with the Canonical-Name. Def-EQ returns the new canonical name assigned to this new class, and will not otherwise return (Resume will unwind to top,

Abort leaves everything alone, and **Continue** will force the union to proceed even with the inconsistency.)[3] **Def-EQ** does NOT set up a new **Undo** structure, as does **Define-Equality**, thus is called from inside the HNAME package by **Union-References**.

Hgenmap *Canonical-Name 1-Complex-Form* &Key *Context*
Returns a list of the fact accessors in the class Canonical-Name that fit the 1-Complex-Form as a template (including any typed rvariables, appropriately). As a second value, returns the bindings needed to generate each entry in the first list. Thus, it will index into the representation of the function to select rows and columns of argument pairs whose values are equivalent to the Canonical-Name - it is not really unification.

Hgenmapall *1-Complex-Form* &Key *Context*
As **Hgenmap**, but it is not constrained to return only forms that are equivalent to some canonical name. Rather, an assoc list of all canonical name accessors as keys (as known for the head function of the 1-Complex-Form) and a list of lists of possible rvariable *bindings that would make the* 1-Complex-Form equivalent to it are returned. NOTE: This function will not work as coded if any of the args in the 1-Complex-Form are themselves canonical names.

Test-Equality *Item1 Item2* &Key *Context*
This function returns a boolean indicating if the two fact or canonical name accessors (Item1 and Item2) are compatible. That is, if the two facts are in the same canonical class, the fact is in the passed canonical class, or the two canonical classes are the same in this context.

The following functions are NOT exported.

Check-Compatibility *Can1 Can2* &Optional (Context *Current-Context*)
Tests that the two canonical classes are compatible, that is, there is no fact alpha s.t. not (alpha) is a member of the other class. Returns NIL if everything ok, otherwise first instance of problem detected.

[3]Actually, this is not yet implemented.

Undo-Current-Equality *Context*
Undoes whatever work we've done so far adding an equality.

Commit-Equality *Context*
Garbage Collects the stuff we left lying around in case of an undo. Specifically, process the undo list, and delete references to old canonical names in fact structures.

Delete-Obsolete-Cnames *Fact-Accessor Cname-Reference*
Take a fact accessor and process it's canonical-name alist: delete obsolete references to Cname-Reference.

12.3 Design Description

Define-Equality basically does some simple argument verification, looks up the canonical name of one of it's arguments (or generates one if *neither has them) and calls* **Def-EQ** on the results. The only other thing this function does is clear out the **Undo** structures from the last call, preparing for a possible failure along the way.

Simplify-Form takes is argument and attempt to translate it into a **Fact** that **Find-Fact** can be called on. This may involve calling itself recursively if one of the arguments in the passed Form involves a **Form** itself. It returns the canonical-name of the fact if it does get translated, and the fact has a canonical name.

Hgenmap and **Hgenmapall** as they are currently implemented are very straightforward, and probably much slower than they need to be. **Hgenmap** gets the set of members of the class Canonical-Name, and then invokes the Unifier's **FN-Term-Unifies-With-Form-P** to check if each member fits the 1-Complex-Form. **Hgenmapall** goes and looks up everything in the KB that has a head that unifies with the passed head in the 1-Complex-Form, and then checks to see if the rest of the item unifies with the 1-Complex-Form as well, and returns it on an alist with its canonical name.

Test-Equality is pretty straightforward. It gets the local accessors for the passed Items, and sees if they have the same canonical name, or are (now) **Eq**.

12.4 Still To Do

- Right now, None of the flags do anything.
- It is unclear that ***Enable-Heq-Warnings*** is useful. There currently aren't any warnings, and I'm not sure when one would generate one, anyway.
- **Hgenmap** and **Hgenmapall** should be expanded to utilize some sort of hashed representation of objects. What needs to happen is that objects that are accessed via these functions will typically have one or more argument unifying with a rvariable and the rest constant. If a pattern can be detected in such access, that is, one or more arguments are predominantly always unified with a rvariable, a hashtable should be set up (and maintained by the HNAME package) which will make such lookup much faster. This can be *done in the current HORNE system* [Allen and Miller, 1986], but it must be done manually by the user. The current implementation works, of course, however, it could be much faster if a reasonable pattern (say, the first argument is always a rvariable) is detected...This would be taken advantage of, of course, by the regular lookup functions used by the prover, not just for the rare calls to these functions.

Chapter 13

The Type Assertion Interface

13.1 Interfacing to the Interface

Only important functions not already described in the user's manual are presented here.

function-table Setqed to the hashtable holding function typing information

Make-Function-Table
Creates a hashtable and binds it to *function-table*

Init-TypeA
initialize the type-assertion system.

Find-Or-Create *name*
Return the rtype for the named type, create a new type if necessary.

Set-RType-Relation *rtype1 rtype2 relation*

Make the two appropriate entries in *TypeKB* for the relation between the two types indicated. Rel[illegible]on is either the name of the intersection, :Subset, :Superset, :Disjoint, :Nondisjoint, or :Equal.

Structured types have several interesting internal functions, though they would not normally be called by the user[1].

Run-Foldable-Constraints *Constraint-Form Instance*

Takes a foldable constraint (a builtin or lispfn with an assert fn, or a Rhet predicate that just needs asserted) with the variable **?self**, binds the variable to the pased value and adds the result.

Process-Roles-of-Instance *Instance Type Roles*

Where Roles is of the form (r1 v1 r2 v2 r3 v3 ...), *for each* (r v), add `[Add-EQ [F-R Irstance] V]`. While we are at it, process constraints and initializations. This is the function that is run when we create an Instance of a Type.

[1]The advanced user may wish to Advise them.

Chapter 14

The Type Database

This subsystem is not fully realized. All of the functions are implemented, but they will only work on simple type expressions, rather than a more complete type calculus we hope to eventually support.

14.1 Overview

The basic algorithm for the type subsystem is fairly simple. Given two simple types, we look in a precomputed two dimensional table, and find their relationship there. Like the equality subsystem (*q.v.*), the equality system has been optimized for fast lookup, and the work done when type relationships are asserted to precompute the table. Since the type table is built offline (the reasoning system not being *monotonic* over changes in the type table), this should not be an issue. *Type reasoning could be implemented more completely and slowly* using the usual horn clause resolution mechanisms, however, the primary idea is to make a limited amount of reasoning about the types of equality terms and rvariables to be quite fast.

Complex type expressions, *e.g.* those involving subtraction, are computed at and reasoned about at run-time. At the moment, the type reasoner is only capable of dealing with type intersection, and type subtraction, and that only in a very simple precedence form. All types ultimately must resolve to basic types (that are stored in the table) and come out to a set of types that are intersected, and a set of types that are subtracted from the intersection to give the result. Thus, *e.g.*, the type **(Human Female - Moron)** would intersect the types Human, and Female (possibly coming up with Women, if that were appropriately assert'ed), and subtracting the Moron type from it. But the similar expression ***(Human - (Female Moron))***, *that is Humans that aren't* Female Morons is not expressible. Instead the system would have to treat it as **(Human - Female Moron)** which given the usual intuitive definitions would be the same as **(Human Male - Moron)** which is quite different since we wanted to only eliminate female morons, and instead eliminated all females and all morons!

14.2 Interface

Some of the following are exported mainly for the use of the TYPEA package. It is unlikely any of the flags other than ***T-Nil-Itype-Struct***, ***T-U-Itype-Struct***, and the other predefined Itype structures will be needed by any other packages and should be considered exclusively for the use of TYPE and TYPEA.

T-NIL-ITYPE-STRUCT An empty type, the least general of all types.

TYPELIST This is a hash table of all the types that have been asserted to the system. Each type has as it's value the index into the typekb table that is it's row and column.

Last-Index (rvariable, initial value of 5) The index of the last type created. (T-U (0) and T-Nil (1) are already there, as are the Lisp related types)

TYPEKB This is the type array. It supports the following relationships:

:equal for two types that are identical.

:superset if for entry [a b] a is a superset of b.

:subset for the opposite.

:intersection if [a b] may intersect, but the intersection is unnamed.

:partition if for entry [a b] b is a subset of a, and for all other subsets of a there is no overlap (part of a cover of a) note that [b a] will be of *type* **:Subset**, *and* **:Partition** *implies superset.*

a non-keyword symbol if [a b] intersect then the intersection has the name of the non-keyword symbol (which is also a type). That is, when we look up the type relationship in the table, if the entry is not a keyword symbol, then it is the type that represents the intersection.

:exclusive if [a b] do not intersect.

:unknown if the relationship is undefined, and could not be derived.

T-U-ITYPE-STRUCT An instance of the universal type, *the most general (:het) type.*

T-ATOM-ITYPE-STRUCT An instance of the type for Lisp atoms (must be in keyword package to be recognized).

T-LIST-ITYPE-STRUCT An instance of the type for Lisp lists.

T-Lisp-ITYPE-STRUCT An instance of the type for any Lisp object.

T-ANYTHING-ITYPE-STRUCT An instance of the type for any *object whatsoever, Lisp or Rhet.*

Exported functions of this package are identical (possibly names changed) to the functionality described for the types system in the Rhet User Manual.

The following are non-exported functions of the TYPE package.

Type-Mode (variable, initially default) **:Default** to assume type with unknown relationship are disjoint; **:Assumption** to assume *they overlap.*

Init-TypeKB
Initialize the type-kb system.

14.3 Remains to be Done

- Do we really need *Typealist*?
- Complex types are not handled; we probably want to expand to a general type calculus at some point.

Chapter 15

The Hierarchical Name Subsystem

Using the package system[1], this subsystem's job is to keep facts about axioms and equalities relative to some context. Facts that are added to a particular context should shadow any facts with the same name (accessor) from a parent context. The request for modification of a structure in a parent context should (logically) cause a copy of that structure to be placed in the current (requested) context's package, and the modification done to this copy (thus keeping the parent's copy intact).

HNAME hides details of how the KB is stored, for example, if facts are stored in an association list, a hash table or an array.

[1]An implementation decision that may change in the future; packages had a better UI than hashtables, and were thus thought easier to debug things in.

15.1 How to Use It

Facts added with the defaultp argument bound to t are only accessible with the same flag binding on access functions. This prevents the Reasoner from seeing default facts if it is not currently doing default reasoning.

HDEBUG If non-nil[2], doesn't complain about some normally illegal things, like destroying the root context. This flag is used generally to indicate that Rhet is in 'debug-mode', and thus it is exported. There are certain places the code will bind it to T to do useful things (like destroy the root context on a **ASSERT:Reset-Rhetorical** call!)

ROOT-CONTEXT The root of all contexts.

DEFAULT-CONTEXT This is the context used for commands and functions that take an optional context argument, if such argument is left unspecified. Contrast to ***Current-Context*** in the Reasoner. The initial value is T's package, the default most global context.

New-Context *Name Parent-Name*
Creates a context with name Name and parent Parent-Name. All names accessible in the parent are also accessible in Name, unless explicitly removed (see **Remove-fact**). The function always returns **T** (errors are handled by exception).

Pop-Context *Name*
Destroys the context named Name. An error is signaled if it is the parent of some other context. The function always returns T.

Contextp *Name*
This will return **Nil** if Name is not a context (as defined by the user), and the **Hier-Context** structure and context if it is.

[2]Currently it is set to T but will not be in a future release.

Context-p *Context*

This will return **Nil** if Context is not a context, and the name of the context if it is (inverse to **Contextp**).

Accessible-Context-P *Context1 Context2*

Returns T if Context1 is accessible from Context2, that is, is equal or a parent of context2.

Destroy-Context *Name*

Like **Pop-Context**, but the named context need not be a leaf. All children are also destroyed. Returns a list of the names of all contexts destroyed.

Contexts &Optional *Root-Package*

Returns a list of the names of all known contexts, in tree form (*i.e.* specifying the inheritance hierarchy). The optional Root-Package should only be used internally by the function (for recursive calls).

Symbol-Context *Symbol*

Returns the context a particular accessor is in.

Context-Parent *Context*

Gets the Context's parent.

Context-Children *Context*

Gets the Context's children.

Convert-Name-To-Context *Name*

Convert the Name of a context to the context it represents.

Convert-Context-To-Name *Context*

Convert the Context passed to the external name it is.

Find-Symbol-In-Context *Symbol Context*

This function is like **Find-Symbol**, but we simulate Context's inheritance. That way, we can encode the rules exactly! So look up Symbol in the Context (package) and then recursively in each parent. Stop with the first one found.

Faccessor-p *Symbol*
Returns non-nil if the passed Symbol is a fact accessor.

Caccessor-p *Symbol*
Returns non-nil if the passed Symbol is a canonical name accessor.

Atomic-FN-Term-P *FN-Term1*
Returns non-nil only if FN-Term1 has no arguments.

Atomic-FNTaccessor-P *FNTAccessor*
Returns non-nil only if FNTAccessor is an accessor for a fact that has no arguments.

Delete-Caccessor *Can-Accessor Context*
Deletes a canonical accessor from the KB; Returns non-nil if something deleted This will allow the canonical name to be GC'd away.

Add-Fact *Fact-Accessor* &Key *Defaultp Marker Context*
Add-Fact adds the fact of the type inherently specified[3] to the current context. Returns the new fact-successo· if successful. The fact is also indexed by the inherent index field.

Add-FN-Term *FN-Term-Accessor*
Add-FN-Term adds the term of the type inherently specified[4] to the root context.

Remove-fact *Fact-Accessor* &Key *Context*
Remove-fact removes the passed fact from the current (or specified) Context. Since it is illegal for

[3]That is, the value of the Type field of the Fact structure which is the value of the passed Fact-Accessor.

[4]That is, the value of the Type field of the FN-Term structure which is the value of the passed FN-Term-Accessor.

particular fact to have more than one type, any inherent type specification is ignored. The fact is appropriately removed from any structures that point to it. If Fact-Accessor is not interned in the current (or specified) context, it is made to appear unbound in the current context, but left in it's normal context. This may involve adding the fact to the current context with a truth value of **:Unbound**. The function returns **T** if successful, **Nil** if the fact was not found. Note: It is illegal to even try to **Remove-fact** a fact that has a canonical name - equalities can only be removed by popping the contexts involved.

Generate-Canonical-Name *FN-Term-Accessor* &Key *Localp Context*
This returns the term's canonical-name accessor in the current (or specified) Context. Should none currently exist, it will create one (in which case the implicit type of the term will be used). The term need not be otherwise interned (*i.e.* by Add-FN-Term), as Generate-canonical-name will happily do so automatically. Setting the Local argument forces **Generate-Canonical-Name** to return the Function-Term's canonical name local to this context, rather than by possible inheritance.

HN-Union *Canonical-Name-Accessor FN-Term-Accessor* &Key *Context*
HN-Union puts the term referenced by the FN-Term-Accessor into the class named by the Canonical-Name-Accessor in the current (or specified) Context. The termneed not have been asserted by **Add-FN-Term**, as this function will happily do so automatically. **HN-Union** always succeeds, and the type and consistency checking must be done by the caller[5]. It returns the canonical-name accessor for the class everything was put into.

HN-Find *Canonical-Name-Accessor* &Key *Context*
Returns all terms in the class named that are accessible to the current (or specified) Context.

Find-Closest-Children-Can-Union *FN-Term1 FN-Term2 Context* &Key *Recursivep*
This function takes two function term accessors and a Context and returns an alist of all child contexts and canonical name accessors in those contexts, if any, one or the other has. The union part is that it will

[5]This keeps type lookup out of this package (mostly), and allows TMS to be exclusively in upper packages. I Hope.

only report the canonical name of ONE of the two, which is what we need to process unions. (we will have to do a recursive union in all contexts that either of these terms are part of a class in.) It will ignore child contexts of other reported children. *i.e.* it returns an alist, where an entry is of the form context.canonical-name An example might be for the simple context inheritance $ROOT- > A- > B- > C$ where term FOO1 has a canonical name in each context, this function evaluated from context A would return ((B.can-name)) rather than ((B.can-name)(C.can-name)) since C is a child of B and so describing B suffices.

Generate-Term-Index *Term*
Returns a Hash-Index for the Term. The Hash-Index will be how the Term will typically be looked up in the Context.

Generate-Form-Index *The-Form*
Returns a Hash-Index suitable for finding all Terms that potentially match the Form.

Unexported constants and functions.

WHO-AM-I String to use to check for the context structure in a package—the name of the symbol we look for.

Print-Fact *Fact-Structure Stream Depth*
Pretty-prints a fact structure. This is supplied as the print routine for the Fact defstruct.

Print-Can *Canonical-Name-Structure Stream Depth*
Pretty-prints a Canonical Name structure. This is supplied as the print routine for the Canonical Name defstruct.

Print-Form *Form-Structure Stream Depth*
Pretty-prints a form structure. This is supplied as the print routine for the Form defstruct.

Print-Var *Rvariable-Structure Stream Depth*
Pretty-prints a rvariable structure. This is supplied as the print routine for the Rvariable defstruct.

Convert-To-Real-Name *Name*

Convert the outside world's view of a context's name to the internal name. The inverse of this function is **Convert-To-Outside-Name**.

Convert-To-Outside-Name *Name*

The inverse of **Convert-To-Real-Name**.

Context-Cleanup *Context*

This function cleans up references to Context in parent contexts. It is called by **Pop-Context-Internal**. It is where we clean up child context references in the canonical name field of a **Term**, for instance, and eventually clean up TMS justifications that refer to children, *etc.*[6].

Blast-Symbol-In-Context *Symbol Context Root-Context*

This function follows the inheritance tree of a context up to the root (as passed) and interns Symbol (passed as a *string* - to make intern work right) in each (with the possible exception of the Root-Context, which will have it already unless it is ***Root-Context***.) It plays the game of setting the value of a particular instance of the Symbol to be the list **(NIL PARENT'S-INSTANCE)**, so updating the parent works the way we want it to. It returns what to set the current symbol to. This function should not be called with Context of ***Root-Context*** since that special case should be handled more simply. (used to determine recursion is ended).

Install-Fact *Symbol Key-Accessor-Function Context*

This function takes the fact accessor (Symbol), or function term accessor we want to install in a context, and a function to apply to the CLTEVAL'd fact to get the data we want to invert on. (the key).

Remove-Fact *Symbol Key-Accessor-Function Context*

This function takes the fact accessor (Symbol) we want to remove from a context, and a function to apply to the **Eval'd** fact to get the data we want to deinvert on. (the key).

[6] And I had so much hoped to keep TMS out of this package! Oh well...

Generate-Reference *FN-Term-Accessor Referand* &Key *Context*

This function takes a FN-Term-Accessor, sees if it has a canonical name, and if not generates one. It then creates a reference to the passed Referand for the canonical name. (the Referand is either a fact accessor or a function term accessor). It returns a list of contexts in which the FN-Term-Accessor has canonical names.

Delete-Reference *FN-Term-Accessor Referand* &Key *Context*

This function is the opposite of **Generate-Reference**, in that it takes a FN-Term-Accessor and a Referand and undoes any reference to the Referand by the term's canonical class.

Union-References *FN-Term-Accessor Can-Name-Accessor Context*

This function takes a FN-Term-Accessor and a Can-Name-Accessor which is local to the passed Context. The FN-Term-Accessor was referenced by a canonical name which is being unioned into the canonical destination name passed. In fact, the FN-Term-Accessor can be a canonical name, rather than some individual term.

Update-Term-With-Can *FN-Term-Accessor Can-Name-Accessor Context*

This function takes a Function Term Accessor and a Can-Name-Accessor which is local to the passed Context. The FN-Term-Accessor was in the set of a canonical name being unioned into the canonical name passed. Therefore, all we need to do is change the canonical name for the current context of the passed term to be the destination Can-Name-Accessor. This adds a new entry for the current Context. Actually, that's not all we need to do - we must update the hashtable for the term as it is indexed by canonical names on the arglist. Note that we leave the hashtable entry for the plain arglist alone. This is only important if we are currently in the context the entry is made in, otherwise **Find-FN-Term** will do the right thing.

Union-Recursive-Class *Con-Can FN-Term-Accessor1 FN-Term-Accessor2*

For some representative term in the canonical class passed, do a union with each child, unless for some reason they are both already in same canonical class in that child. (check) Trick is that we don't know which term canonical name passed is associated with up front. Con-Can is a cons whose Car is the context, and whose Cdr is the canonical-name accessor.

Update-Hashtable-With-Can <*Fact-Accessor or FN-Term-Accessor*> *Context*
Called when any argument of the fact or term has changed canonical classes. We need to reevaluate the arglist for updated canonical names in the passed context, and update the term's hashtable in this context for them. This may imply that this is the first hashtable for the head in this context, so we may have to do some linking...Note: Fact-Accessor better be the right accessor for the current context, we don't check.

15.2 When Not to Use It

- Higher-level interfaces for equality should be used, rather than attempting to use HN-Union directly.

15.3 How It Does It

First, contexts are implemented on the package system. The reason for this is twofold: packages (at least on the Symbolics^{TM} and Explorer^{TM}s) are very efficiently implemented as hashtables. The advantage to using packages over hashtables themselves is debugging: it's a lot easier to be able to type the internal name of the context as a package name and the accessor and see what's there, rather than having to always look things up more or less manually in a hashtable.

The context related functions take a context name from the user and prepend "Hier-" to it it come up with the package name, and creates a package which has no :USE list. This is done to keep the system from complaining about multiple definitions, and making us shadow things explicitly, *etc.* (and it will still complain anyway, even when things are unambiguous, just to make sure you are aware there are two symbols with the same name!) So, the function Find-Symbol-In-Context does the inheritance for us. Using the WHO-AM-I constant, it can recursively look for a symbol in each parent if it doesn't find it in the passed context.

The "T" context is considered the most general one. Thus it is handled a bit specially by the functions: it's package is made the value of ***Root-Context*** which is often checked to see if we are done looking for something, and there is no further to look.

The **Canonical-Name** slot in a **FN-Term** structure contains an alist whose keys are contexts, and whose values are canonical-name accessors. This is to allow for a term to have a canonical name relative to the context. Thus, given that I am in context FOO which is a subcontext of T, and I attempt to lookup BAR via it's accessor **Accessor-00023** I will find it, perhaps, in context T, but be able to get it's canonical name in BAR distinct from it's one in T.

Here is a minor idiosyncrasy it pays to be aware of: If you generate a canonical name for a term that the system does not know about in some context, and then again generate it in a parent context, the system assumes you are dealing with two DIFFERENT instantiations of the same term. That is, there will be two terms interned, in the two packages, which will have the same accessor. If the assertion order were reversed it would be the SAME term, with two different canonical names. This makes a difference when certain indirect or direct unions happen: In the first case, with two different facts, if we then do a union in the parent, no union is done in the child - it is a different fact. In the second, the more obvious thing happens.[7]

This makes sense when you consider that the user and system in some sense isn't supposed to "know" in a parent context about the contents of child contexts - thus the first example creates a new fact, because it shouldn't be the same as the child contexts fact - if it were it would be a case of the child accessing the fact in the parent which is what happens in the second example - the fact has to be present in the parent before the child can access it.

The Type of an object can also vary with the context: consider the contexts above, where in the root context we know that Tweety is a bird and that penguins cannot fly, and that a complete partition of bird is penguins and non-penguins. If in FOO we have another object Fred the penguin and we discover that Fred and Tweety are the same (so we make them equal) the type of Tweety is compatibly restricted to penguin. But Tweety's type in the root context cannot change, since the equality isn't present there (in fact Fred may

[7]This may no longer be a problem now that the treatment of facts and function terms has been formally separated.

not exist in the root — he is only in the FOO context -- perhaps he is only a hypothetical entity). What we do is if a term has a canonical name we get it's type from the Canonical structure, which varies by context. If it doesn't we use the type declared in the FN-Term structure instead.

Canonical names are in some sense the most complex abstract item implemented by the HNAME package. The main reason is that they must be kept straight depending on context, and before and after unions are done. A particular term accessor may have canonical names in several contexts as mentioned above, which is useful to determine when additional work must be done in a child context. That is, given that some function term, say [F A] is interned in the root context, and it is unioned with term [F C] in subcontext FOO1. If in the root we want to union [F A] with [G] we have to 'remember' somehow to update the union in the child context as well. It is not enough to make these things inherit: consider [A] and [B] in the root and [C] and [D] in the child with [EQ A C] and [EQ B D] (again in the child). Unioning A and B in the root would give us problems if we used inheritance – looking up the canonical name of B might give us B and A and C but require extra work to find D. So instead we just use these canonical names stored in the term accessor in the most root-ward copy of the term (which we get via the functions Find-Closest-Children-Can-Union and Find-All-Children-Can) to tell us which child contexts will also require a union. We do the union in the child before we finish the one in the parent, because we don't want to give the child conflicting information via inheritance were we to do it in the parent first. The Union-Recursive-Class function is used to do this work.

At that point we decide which of the two canonical classes we are going to union together we will copy to the new canonical name and which we will process in. (Copying is easy, processing involves updating references). We choose the class with the most references to be copied. We update the term structures with the new canonical names using Update-Term-With-Can and call Union-References to do the dirty work with the references.

Getting the canonical name from a function term basically involves either finding the canonical name associated with the current context, or calling Get-Canonical-From-FN-Term recursively on the parent context until we do find it in the alist (or fail). Facts, themselves, have no canonical names.

The dirty work of Union-References requires us to combine the reference fields of the two canonical classes HN-Union was called on. One set of references was simply moved to a new canonical name (the destination), the other must be processed in. Therefore, first check to see if there is an equivalent in the destination context's references for each referenced item, given that the canonical name accessor contains all equivalences of arguments necessary. That is, if we had [F A] and [F B] and now are unioning A and B, (A ΣΣ B's class) the passed FN-Term-Accessor will be for [F A], the passed Can-Name-Accessor will be for B's class including A but only referencing [F B] plus anything else that was originally there. We want to detect that [F A] and [F B] are now equivalent, and *do the recursive union*. *If they are not, the reference can just be added* as a reference for Can-Name-Accessor. This involves checking the canonical names of the arguments to function terms with matching heads - we use **Can-Atom-Equivalent-P** to make sure the heads are equal, and just check that the canonical names of each argument in this context are **Eq** to each other.

The basics of adding facts in the first place is relatively straightforward: The fact accessor is interned in the appropriate package (context) and the head is also separately interned. A property on the head **:Hash-All-Args** is *created and set to a hashtable* whose keys will be the arglists of facts with this head and whose terms will the the fact accessors. The fact is also interned by its index and its marker fields. Each of these (including the head) have as their value a list of all accessors with this head, index, or marker. The last cdr of the list in this context, so to speak, is a pointer to the list in the parent context. In this manner, we can add facts with the same head to a parent context, and still pick it up from the head accessor in the child - it will appear to be on its list! The hashtables are not so updated. Normally finding a fact involves recursive lookups in the hashtable for each head in successive parents until an accessor is found. The accessor is then returned unless it's truth value is **:Unbound** in which case the fact is considered to have been deleted and treated as if not found. (The **:Unbound** value is created by **Delete-Fact** when the fact accessor was added to some context, and deleted from a child - thus we still want to be able to access it from the parents of this child. We reintern the fact in the child context and set it's truth value to **:Unbound** which will hide it from this child and all of its children.)

The only other important thing to know is that each fact that is present on an arglist is given a canonical name, and the fact itself made a reference of this argument (via **Generate-Reference**). Since often there will be more than one argument equivalent to the one when the fact was added (at least eventually) every time we union something which has a reference, we update the hashtable (as attached to **:Hash-All-Args**) and also use the canonical names of the arguments. This can get us into trouble, if we, say, union two facts such that their hashtable function would collide (*i.e.* on the same arglist we could get more than one fact). In this case rather than a fact accessor, we put the canonical name accessor into the value of the hashtable.

15.4 Still To Do...

- Several functions don't take advantage of hashtables (or more importantly, indexing and hashtables), and probably should.

- While we don't allow user specified hashing for purposes of unification (*i.e.* having the user specify which argument term of a function term will likely be matched on so hash on it, others being constant), right now we don't do much of anything automatically either. This makes things slower than they need to be, but upgrading it isn't a high priority either. *Functionality before Performance!*

- Too many functions take a fact and look up the canonical name, or take a canonical name, and have to double check that it's the right one for the context. Hashtables usually store the fact as the object of the hash, since storing the cname forces an entry into each context. Sometimes we get an entry in each context anyway, which isn't strictly needed. This all needs to be cleaned up for greater efficiency. The trade off is putting the cname in the hash, and have more entries, and possibly losing information when attempting to do unifications. The actual pattern may matter, since [P A] may be asserted to be Eq to [F T] and thus should unify, (will be in the same cano..ical class) but that's not an excuse for [F ?x] to unify with [P A]: we have potentially lost information if F and A are distinct. That is, they should unify, but with different internal state as it were: ?x is bound to T.

- Facts, Forms, *etc.* are all Defstructs. It would have been a lot nicer (and made the code simpler) if they had been flavors instead, since a lot of code that tests to see which one they are and get the (same) slot could have been simplified. Theoretically a sort of hierarchical Defstruct as we did for axioms would have cleaned things up a bit too, but only the obvious stuff. If CLOS gets off the ground though, and is supported fully by the various manufacturers, maybe we can rewrite stuff to use it. As it is, this was something we sacrificed for CL compatibility.

Appendix A

Installation of Rhet version 15.25

Installation is typical for systems distributed for Symbolics or Explorer hosts, *e.g.* there is a Defsystem file, and appropriate system and translation files should be installed in the site directory. Non-Lispm hosts should see the Porting Rhet appendix.

There is, however, one installation option, which determines how Rhet installs it's readtable. The parameter UI:***Make-Rhet-Readtable-Global*** determines whether the readtable on the machine is replaced, or if only the Rhet process itself will use the Rhet readtable. If this term is not **Nil**, the symbol ***Readtable*** is **Setf**'d to the Rhet readtable, while if this value is **Nil**, the Rhet process (and the editor mode) merely binds it. Note that if this latter option is chosen, care must be taken with attempting to read rhet files in from a Lisp listener, or programmatic interface with Rhet, since these processes will need to first bind their ***Readtable***s as well. Last, it is a function of the **USER:*Rhet-Initializations*** list, which is normally run immediately after loading Rhet, to set up this readtable. See section B.2, for more details on initialization lists.

A.1 Special Instructions based on machine type

A.1.1 Symbolics

Rhet should load and run on a Symbolics "out of the box" if your distribution was via a "Distribution Tape". TeX (available from SLUG) is also required to use the distributed defsystem. If you do not choose to get TeX, you may delete references to the manual from the defsystem, and than Rhet should be able to load. (The machine really only needs to know about TeX file types, rather than have the entire TeX distribution).

A.1.2 Explorer

You may need to delete italicized expressions from the sources (which are only used in the Rhet experimental version), though it should not be strictly necessary. Still, if you wish to report a bug in Rhet, we cannot reproduce anything unless it is from a released version. The Explorer also needs to know about TeX file types.

Note that you must load the G7C system (provided in the public directory in releases beginning with 4.1) in order to load Rhet.

Certain other public domain subsystems may be loaded by the Rhet defsystem file; these will be found in the utilities subdirectory.

Appendix B

Porting Rhet

This section is vastly underspecified. As CL is extended with error handling, objects, etc. divergences from the standard should become less necessary, and as compiler technology improves, even less so (efficiency hacks will no longer be needed).

B.1 Overall comments

Note that the source code is occasionally multi-fonted (typically using the Symbolics format). This may cause problems on non-lispms (or even non-Symbolics machines). Font information will appear to ASCII devices to be a 006 (binary) character[1], followed by other font-specific information. For a good example of this, see the copyright.text file in the home directory of the distribution. On a Symbolics, the copyright line appears as:

```
    ;;; © Copyright (C) 1988, 1987, 1986 by the University of Rochester.  All rights
reserved.
```

[1]In ASCII, this is control-F or "ack".

For the most part, font information appears inside of comments and should not effect compilation. Usage of non-comment font information to date has been very restricted, and occurs rarely:

- Source that appears in *italics* indicates that it is not part of the current released version. It is not part of any patch on the version, but is compiled in experimental versions only. This serves as a reminder to the maintainers that to patch a function or definition with italicized code, other code may have to be patched as well, or the italicized code must be deleted. Commented out source code that is italicized is *PRESENT* in the released version but will be absent in the exprimental version. Normally, this is because other code has replaced it. If you are porting Rhet or otherwise compiling from the sources rather than using binaries and patches as distributed (non-lispm environs), you may simply delete all italicized code not in comments, and add all italicized code in comments, OR delete the italizing characters from the file, and use the experimental versions. Typically, this usage has been to add fields to structures that are widely used: the patch files would be so large that the system should be recompiled as a matter of principle, but either the additional supported funcitonality needs further testing before official release, or the release process itself was infeasible at the time for that change.

B.2 Initialization Lists

Rhet takes advantage of the Lispm initializations lists to defer certain processing until after all of Rhet is loaded. Such forms are put on the **USER:*Rhet-Initializations*** list and are expected to be executed only once, immediately after loading Rhet. Other forms may also be put on the system's cold or warm boot list. The cold boot list is expected to be executed after a cold boot, while the warm boot list is executed after every warm boot and all cold boots. Forms are added to these initialization lists via the **Add-Initialization** function, and run using the **Initializations** function.

B.3 Package Handling Functions

Rhet makes use of some package handling functions available on the lisp machines, that are possibly implementation dependant (they are not in basic common-lisp). Specifically:

Pkg-Kill This deletes a package from the system and uninterns any symbols that had that package as their home package.

Mapatoms A mapping form that maps a function over all of the symbols in a package.

B.4 High Level User Interface

Virtually everything in the window-interface directories are extremely implementation dependant, as they depend on the host machine's model of processes, windows, the mouse or other i/o interface, prompting for typed expressions, documentation and assistance presentation, *etc.*. Rhet can be loaded and run without these functions, but a complete implementation should have equivalent functionality running on the host machine to facilitate rapid prototyping and debugging of systems implemented on Rhet.

Bibliography

[Allen and Miller, 1986] James F. Allen and Bradford W. Miller, "The HORNE Reasoning System in COMMON LISP," Technical Report 126, University of Rochester, Computer Science Department, August 1986, Revised.

[Allen and Miller, 1989] James F. Allen and Bradford W. Miller, "The Rhetorical Knowledge Representation System: A User's Guide," Technical Report 238 (rerevised), University of Rochester, Computer Science Department, March 1989.

[Bruynooghe and Pereira, 1984] M. Bruynooghe and L. M. Pereira, "Deduction Revision by Intelligent Backtracking," In J. A. Campbell, editor, *Implementations of Prolog*. Ellis Horwood Limited, 1984.

[Bruynooghe, 1982] Maurice Bruynooghe, "The Memory Management of PROLOG Implementations," In K. L. Clark and S.A. Tarnlund, editors, *Logic Programming*. Academic Press, 1982.

[Cox, 1984] P. T. Cox, "Finding Backtrack Points for Intelligent Backtracking," In J. A. Campbell, editor, *Implementations of Prolog*. Ellis Horwood Limited, 1984.

[Fagin, 1984] Barry Fagin, "Issues in Caching Prolog Goals," Technical Report UCB/CSD 84/204, University of California at Berkeley, Computer Science Division, November 1984.

[Haridi and Sahlin, 1984] S. Haridi and D. Sahlin, "Efficient implementation of unification of cyclic structures," In J. A. Campbell, editor, *Implementations of Prolog*. Ellis Horwood Limited, 1984.

[Hopcroft and D., 1979] John E. Hopcroft and Ullman Jeffrey D., *Introduction to Automata Theory Languages, and Computation*, Addison-Wesley, Reading, Massachusetts, 1979.

[Kahn and Carlsson, 1984] K. M. Kahn and M. Carlsson, "How to Implement PROLOG on a Lisp Machine," In J. A. Campbell, editor, *Implementations of Prolog*. Ellis Horwood Limited, 1984.

[Kornfeld, 1983] W. A. Kornfeld, "Equality for Prolog," In *Proceedings. 8th IJCAI*, Karlsruhe, W. Germany, August 1983.

[Mellish, 1982] C. S. Mellish, "An Alternative to Structure Sharing in the Implementation of a PROLOG Interpreter," In K. L. Clark and S.A. Tarnlund, *editors*, *Logic Programming*. Academic Press, 1982.

[Sterling and Shapiro, 1986] Leon Sterling and Ehud Shapiro, *The Art of Prolog: Advanced Programming Techniques*, MIT Press Series in Logic Programming. MIT Press, Cambridge, Massachusetts, 1986.

[Warren, 1977a] David H. D. Warren, "Implementing PROLOG — Compiling Predicate Logic Programs: Volume 1," Technical Report 39, Department of Artificial Intelligence, University of Edinburgh, May 1977.

[Warren, 1977b] David H. D. Warren, "Implementing PROLOG — Compiling Predicate Logic Programs: Volume 2," Technical Report 40, Department of Artificial Intelligence, University of Edinburgh, May 1977.

[Warren, 1986] David H. D. Warren, "Optimizing Tail Recursion in Prolog," In Michal van Caneghem and David H. D. Warren, editors, *Logic Programming and its Applications*. Ablex Publishing Corporation, Norwood, New Jersey, 1986.

[Wilensky, 1986] Robert Wilensky, *Common Lispcraft*, W. W. Norton and Company, New York, 1986.

Index